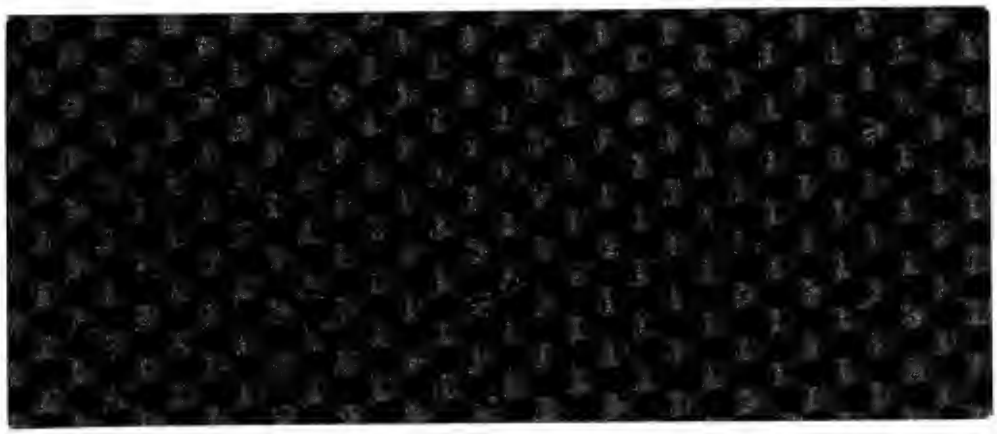
# An Evaluation of Monitoring Algorithms for Access Anomaly Detection

by

*Anne Dinning (dinning@cs.nyu.edu)* †
*Edith Schonberg* ‡

Ultracomputer Note #163
July, 1989



# An Evaluation of Monitoring Algorithms for Access Anomaly Detection

by

*Anne Dinning (dinning@cs.nyu.edu)* †
*Edith Schonberg* ‡

Ultracomputer Note #163
July, 1989

†This research was supported in part by an IBM Graduate Fellowship.

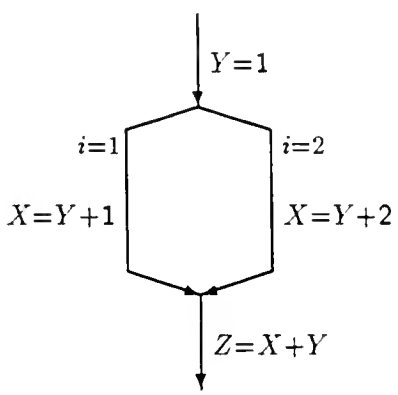
‡This research was supported in part by the Applied Mathematical Science subprogram of the office of Energy Research, U.S. Department of Energy under contract number DE-FG02-88ER25052.

*Abstract*

One of the major disadvantages of parallel programming with shared memory is the nondeterministic behavior caused by uncoordinated access to shared variables, known as *access anomalies*. Monitoring program execution to detect access anomalies is a promising and relatively unexplored approach to this problem. We present a new algorithm, referred to as *task recycling*, for detecting anomalies, and compare it with two other monitoring algorithms. Both analytic and empirical results indicate that task recycling is more efficient in terms of space requirements and performance. Significant conclusions are: (i) Space requirements are in the worst case bounded by $O(T \times V)$, where $T$ is the maximum number of threads that may potentially execute in parallel and $V$ is the number of variable monitored. For typical programs, however, space requirements are on average $O(V)$. (ii) Most of the actual performance costs for monitoring is at synchronization operations - the cost per variable access is a small constant. (iii) The general approach of monitoring to detect access anomalies is practical. Various implementation optimizations, as well as the use of complementary tools, can enhance monitoring to effectively solve the access anomaly problem.

# 1 Introduction

Erroneous non-deterministic behavior in shared memory parallel programs is often the result of *access anomalies*. An access anomaly occurs when two concurrent execution threads access the same memory location in an "unsafe" manner: more specifically, when either two concurrent threads both write or one reads and one writes a shared memory location without coordinating these accesses. The program segment in Figure 1 illustrates an access anomaly. The *doall* construct creates two parallel threads that both write the variable $X$. The value of




```
Y = 1
doall i = 1 to 2
    X = Y + i
endall
Z = X + Y
```

Figure 1: Simple Program With an Access Anomaly

$X$ subsequently used to calculate $Z$ depends on whether iterate $i = 1$ or $i = 2$ writes $X$ last. Therefore the two writes to $X$ are anomalous. Variable $Y$ is also accessed in both concurrent threads $i = 1$ and $i = 2$; however, this is not an anomaly because both accesses are reads. Similarly, the assignment to $Y$ in the first statement does not cause an anomaly because this write is always performed before either read.

An access anomaly is a specific instance of a general class of bugs in parallel programs known as *race conditions*. Typically access anomalies result from incorrect inter-thread coordination, such as missing *lock* operations, or program logic errors, such as incorrect array referencing. While certain *internal non-determinism* [7] does not affect the outcome of programs and is safe (e.g. the non-deterministic order in which a lock is granted), access anomalies often introduce *external non-determinism* that modifies execution results. Although some programs are designed to contain access anomalies, this type of external non-determinism is usually unintentional.

Traditional debugging techniques are of limited use in finding errors caused by access anomalies because such bugs are sensitive to timing. Alternative approaches that have been proposed for automatic detection of anomalies fall into four major classifications, two of which are static techniques and two of which are dynamic. Static analysis [7,4] builds upon classic data dependence and flow analysis to identify potential anomalies prior to execution. Symbolic execution [2,16] finds potential concurrent execution sequences by exhaustively searching program control paths, also prior to execution. Post-mortem analysis [1,6,11] is a dynamic method that examines a trace of the accesses to shared memory taken from a single *execution instance* to determine whether anomalies are present. Finally on-the-fly anomaly detection [13,14,15], also a dynamic method, monitors a program while it executes and detects anomalies as they occur.

Dynamic methods are necessary as a complement to static methods [1,7]. Both static analysis and symbolic execution are too conservative, and therefore can fail to isolate real anomalies from those incorrectly identified. Furthermore, it is difficult to perform accurate data dependence analysis on languages that support pointers and aliasing. Even when static analysis fails, dynamic methods pinpoint anomalies precisely.

This paper examines methods for on-the-fly access anomaly detection. The primary advantages of on-the-fly detection over post-mortem analysis derive from data compression. The execution graph is not explicitly built, and shared memory events that are no longer relevant during execution are discarded. Post-mortem analysis requires trace data of size $O(E)$ and time $O(E^2)$, where $E$ is a function of the program execution length. On-the-fly detection can be performed in time $O(E)$ and space independent of $E$. Indeed, if on-the-fly detection is efficient enough, monitoring may be performed continuously without adversely affecting program performance; an access anomaly error may be discovered during program execution in much the same way as array subscript out-of-range exceptions are currently reported. As soon as an anomaly is detected, an exception is raised specifying the variable involved as well as the instruction counter(s). The ability to monitor programs continuously mitigates single execution instance problems, including the problem of probe effects. The transparency of continuous monitoring is further enhanced when static analysis is used to reduce the number of variables that need to be monitored in a single execution. Testing methodologies may also be applied to span the range of possible program executions needed to validate the program.

On-the-fly detection methods are described for parallel programs under two different models of concurrency: the pure fork-join model and a model which incorporates synchronous and asynchronous coordination. We present a new on-the-fly algorithm, called *task recycling*[1], and compare it to two other algorithms: English-Hebrew labeling [12,13] and merging [14,15]. We argue analytically as well as empirically that task recycling is an improvement over the other two algorithms both in terms of space and computational requirements. The merge algorithm has an advantage only when shared variables have multiple concurrent readers. However, em-

[1]This algorithm relies on version numbers, which are similarly used in [6] for post-mortem analysis.

pirical measurements of scientific programs indicate that the number of concurrent readers is very small and on average independent of the degree of parallelism. The task recycling algorithm, on the other hand, is significantly more space efficient than the English-Hebrew labeling algorithm and extends more easily to handle coordination than both of the other algorithms. Various environment-dependent optimizations further improve the performance of task recycling.

Section 2 defines the basic programming model and introduces the partial order execution graph which forms the basis for concurrency analysis. Initially we restrict our attention to the pure fork-join model of execution which is encapsulated in series-parallel execution graphs, and Section 3 describes the three on-the-fly detection algorithms given this basic programming model. In Section 4 we extend this programming model to encompass more general partial order graphs. Section 5 discusses extensions to the three algorithms to handle this wider class of parallel programs. Section 6 presents empirical results for monitoring several scientific benchmark parallel programs using the English-Hebrew labeling and task recycling algorithms. Section 7 concludes with a discussion of more general application of on-the-fly analysis. The Appendix contains the proofs of several properties of partial order execution graphs.

## 2 Problem Definition and Programming Model

The notion of access anomaly was first defined in terms of *Read* and *Write* sets by Bernstein [3]. Each sequence of instructions in a parallel program has a Read and Write set associated with it: the Read set is those variables that are read in the instruction sequence and the Write set is those variables that are written. If two potentially concurrent sequences $S_i$ and $S_j$ meet the following three conditions (known as *Bernstein's conditions*), all execution orderings of $S_i$ and $S_j$ are guaranteed to produce the same result (i.e. $S_i$ and $S_j$ do not introduce any external non-determinism):

$$Read(S_i) \cap Write(S_j) = \emptyset$$
$$Write(S_i) \cap Read(S_j) = \emptyset$$
$$Write(S_i) \cap Write(S_j) = \emptyset$$

If these conditions are not met, concurrent execution of $S_i$ and $S_j$ may lead to access anomalies that alter the result of their execution. In order to evaluate Bernstein's conditions, two types of information must be available: (i) which instruction sequences potentially execute concurrently, and (ii) which variables are accessed by each instruction sequence. The goal of on-the-fly analysis is to maximize efficiency by storing at any time the least amount of information that will be needed in the future.

Our basic model of parallel execution supports closed, nestable fork-join constructs. Concurrent threads are created by a *fork* operation and terminated by a corresponding *join* operation. This basic construct is realized in several common parallel language constructs such as *doall-endall*, which creates a set of homogeneous tasks, and *parallel case*, which creates a set of heterogeneous tasks. A *block* is an instruction sequence, executed by a single thread, that does not include either fork or join operations. (Hence a thread is made up of a sequence of blocks that reflect its interaction with other threads.) We represent the concurrency relationship among blocks by a directed *partial order execution graph* [10]. A partial order execution graph captures Lamport's "happens before" relation and imposes a partial order on the set of blocks that make up an execution instance. The edges of a partial order execution graph represent blocks, the vertices represent fork or join points and are created as follows:

- A *fork vertex* has one in edge (the parent block) and $f$ out edges (the $f$ children blocks).
- A *join vertex* has one out edge (the child block) and $f$ in edges (the $f$ parent blocks).

Initially we do not consider orderings imposed by other inter-thread synchronization, such as events, barriers, and message passing; this restriction is removed in Section 4. Because this basic program model is limited to fork-join constructs, all partial order execution graph are series-parallel graphs.

A partial order execution graph is used to determine whether any two blocks in an execution instance can potentially execute concurrently. This evaluation is made independent of the number and relative execution speeds of processors executing the program. A block $b_j$ is an *ancestor* of a block $b_i$ if there is a path from $b_j$ to $b_i$ in the graph (likewise $b_i$ is a *descendent* of $b_j$). Two blocks are *concurrent* iff neither is an ancestor of the other. This definition of concurrent agrees with the common concept of concurrency:

**Theorem 1:** Two blocks $b_i$ and $b_j$ in a partial order execution graph $G$ can execute in parallel iff $b_i$ is neither an ancestor nor a descendant of $b_j$ in $G$.

We define the *maximum concurrency* of a partial order execution graph to be the maximum number of mutually concurrent blocks. To illustrate these definitions, consider the partial order execution graph in Figure 2 which represents a program segment containing two nested *doall* constructs. This graph has maximum concurrency of 6. Block $b_1$ is concurrent with

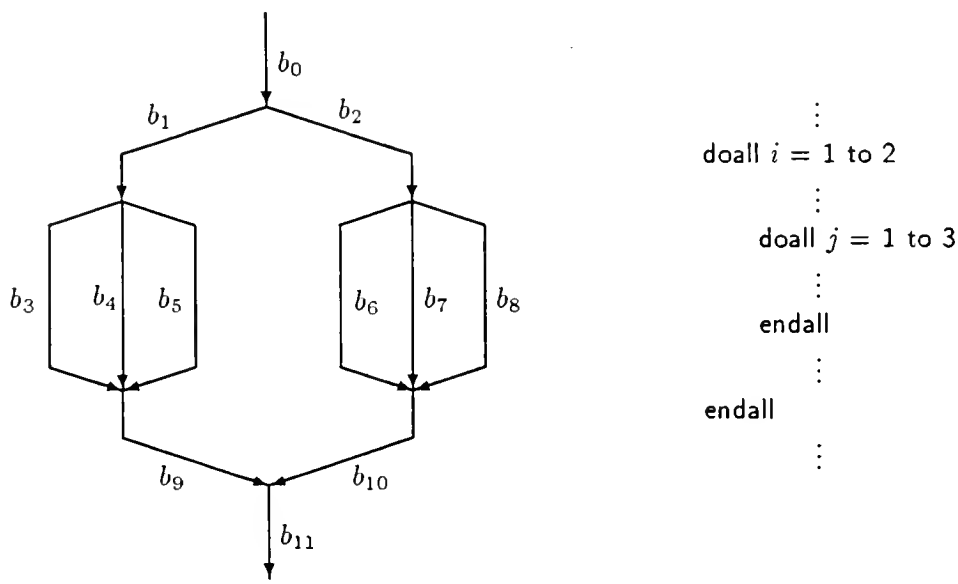



Figure 2: Partial Order Execution Graph for Nested Doall Program

blocks $b_2, b_6, b_7, b_8$ and $b_{10}$; it is not concurrent with block $b_0$ (which is an ancestor) or with blocks $b_3, b_4, b_5, b_9$ or $b_{11}$ (which are all descendents).

In addition to determining concurrency relationships, an access anomaly detection algorithm must ascertain which variables are accessed by each block. A trace of variable access data is in general proportional to the size of the execution graph. However, the on-the-fly algorithms described below compress this data so that storage requirements depend on the maximum concurrency rather than the number of blocks in the execution instance. Unfortunately certain information may be lost because of data compression. In particular, if there are

multiple anomalies involving the same variable, some of them may not be reported. However, at least one anomaly is guaranteed to be reported for every variable which is accessed in an "unsafe" manner.

Read and Write set information can be maintained in several ways. The merge algorithm maintains explicit Read and Write sets for each block; these sets are checked against the Read and Write sets of concurrent blocks at block termination. The *merge* data compression heuristic [14] decreases the total number of Read-Write sets. In the Hebrew-English labeling and task recycling algorithms, the Read and Write sets are maintained in inverse form; the history of accesses to a variable $X$ are stored with $X$ in an *access history*. An access history consists of a list of reader blocks and a list of writer blocks; every time variable $X$ is read or written, its access history is examined to determine whether the current event conflicts with a previous one. Access histories are compressed using a technique called *subtraction* [14] which reduces the size of the reader and writer sets. Both subtraction and merging decrease the number of checks performed as well as the size of the access information stored.

A primary difference among the three algorithms is the method used to determine whether two blocks are concurrent. The next section describes and evaluates these algorithms by comparing the cost of concurrency determination and storage requirements.

# 3 Methods for Detecting Access Anomalies

Given the preceding problem definition and basic fork-join program model and series-parallel partial order execution graph, this section presents three algorithms for detecting access anomalies. We begin by describing the merge algorithm [14,15], which differs more sharply from the other two in that it uses explicit Read-Write sets. English-Hebrew labeling and task recycling, described later, are more similar in that they both use access histories. Throughout subsequent sections, the following notation is used to discuss relative space and time costs:

T - the maximum concurrency
B - the number of blocks
N - the maximum level of nesting of fork-join constructs
V - the number of monitored variables
A - the average number of variables accessed per block
R - the average number of read events in an access history

In the worst case, $R$ approaches $T$ and $A$ approaches $V$. The relative efficiency of the three algorithms depends on the values of these parameters for the specific program being monitored.

## 3.1 Merge Algorithm

The merge algorithm for series-parallel graphs is derived from the following observation: if more than one concurrent sibling block in a fork-join construct read the same variable - thereby creating multiple read events - these events may be collapsed into a single event *after the join operation* [15]. The effect of merging is shown in Figure 3. The variable $X$ is read in three concurrent blocks: $b_3$, $b_4$ and $b_5$. Three distinct records of these read events are stored in the Read sets associated with $b_3$, $b_4$, and $b_5$. However, once $b_3$, $b_4$, and $b_5$ terminate, any subsequent write that conflicts with one of the three read events will conflict with all of them, so that the read events may be merged at termination. This merged record for $X$ is in the initial Read set of $b_9$. The conflict with the write in $b_{10}$ is detected when $b_9$ and $b_{10}$ terminate.

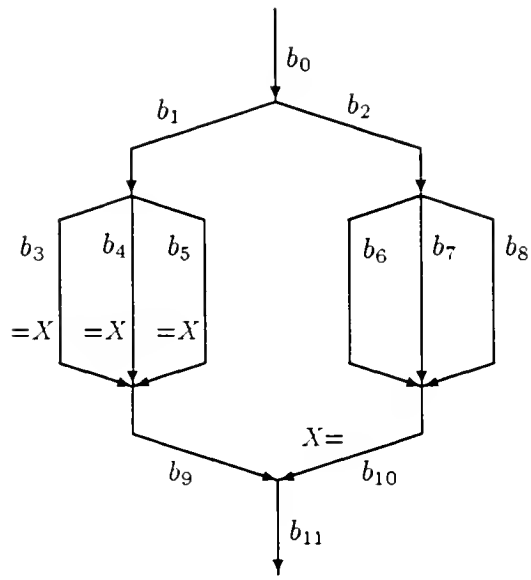



Figure 3: A Partial Order Execution Graph for Merge Algorithm

Note that merging cannot be performed before the join operation since a subsequent write in either $b_3$, $b_4$, or $b_5$ conflicts with only two out of the three read events.

More precisely, Read and Write sets are maintained explicitly as described in Section 2. If block $b$ executes a fork operation creating blocks $b_{c_1} \ldots b_{c_f}$, the Read and Write sets associated with $b$ are preserved, and $f$ (initially empty) Read and Write sets are created to store access events for $b_{c_1} \ldots b_{c_f}$. When the corresponding join is executed, Bernstein's conditions are evaluation to see if any anomalies exist. The $f$ Read and Write sets are compared by performing $f-1$ set intersection operations. These Read and Write sets are then merged with the Read and Write sets for $b$ by performing $f$ set union operations. Hence events performed by blocks which are "horizontally" related in the partial order execution graph are combined. The merged Read and Write set is the initial Read and Write set of the block executed after the join operation. The merge strategy delays Read and Write set comparisons, but eventually every Read and Write set is compared against the Read and Write set of every block with which it is concurrent. For example, when $b_9$ and $b_{10}$ terminate and compare their Read and Write sets, the Read and Write set of blocks $b_1 \ldots b_8$ are implicitly compared as well. Table 1 shows the initial state of Read and Write sets for the blocks in Figure 3.

| *Blocks* | *Initialization* |
|---|---|
| $b_1$ - $b_8$ | $\emptyset$ |
| $b_9$ | $\mathrm{RW}(b_1) \cup \mathrm{RW}(b_3) \cup \mathrm{RW}(b_4) \cup \mathrm{RW}(b_5)$ |
| $b_{10}$ | $\mathrm{RW}(b_2) \cup \mathrm{RW}(b_6) \cup \mathrm{RW}(b_7) \cup \mathrm{RW}(b_8)$ |
| $b_{11}$ | $\mathrm{RW}(b_0) \cup \mathrm{RW}(b_9) \cup \mathrm{RW}(b_{10})$ |

Table 1: Table of Read and Write Set Pair Initial Values

If Read and Write sets are stored as fixed length structures, each Read and Write set requires $O(V)$ space, there is constant work per variable access, and $O(V)$ work at block completion (the union and intersection operation each requires $O(V)$ work). If a variable

length structure is used, the sizes of Read and Write sets grows towards V in proportion to the number of nested fork operations. The number of Read and Write sets required is equal to the number of currently executing tasks plus the number of outstanding open fork operations. Assuming that each fork operation has at least two children, there can be at most $\frac{T}{2}$ outstanding fork operations and therefore $O(T)$ Read and Write sets are required in the worst case.

The merge algorithm has several strengths. The specific properties of series parallel graphs are exploited to determine concurrency, so that no explicit information about the partial order execution graph needs to be maintained. Moreover, because Read and Write sets are private to a block, no locking is necessary when these sets are updated and they may be cached in local memory. Finally, it is easy to discard Read and Write sets at serial execution points; no previous events are relevant once a serial section is reached. Merging guarantees that all variables with at least one access anomaly are detected. However, the delayed checking and compression make it difficult to locate the conflicting accesses precisely and hence diagnostic precision is lost.

### 3.2 Access History Algorithms

In both the English-Hebrew labeling and the task recycling algorithms, an access history is associated with each monitored variable $X$. Every time $X$ is accessed, the access history is examined for possible anomalies and then updated. Therefore, the efficiency of these algorithms depends primarily on how quickly each entry in the access history can be examined and how many entries an access history contains.

To illustrate the use of access histories, consider variable $X$ in Figure 4. In this example,

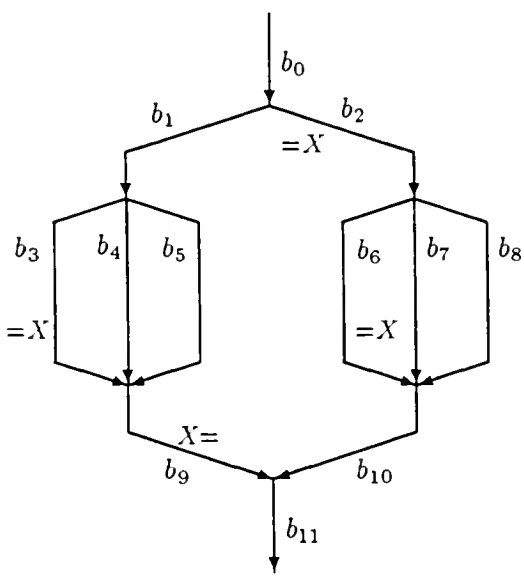



Figure 4: A Partial Order Execution Graph with Access Anomalies

the write of $X$ in $b_9$ conflicts with the reads of $X$ in $b_2$ and $b_6$. Suppose $X$ is read in $b_2$, $b_6$, and $b_3$ in that order, and then written in $b_9$. After the first read, the access history of $X$ contains $b_2$. When $X$ is next read in $b_6$, $b_6$ is added to the access history and $b_2$ is deleted. The $b_2$ read event is no longer needed: any subsequent write event that conflicts with $b_6$ also

conflicts with $b_2$. On the other hand, when $b_3$ is added to the access history, $b_6$ cannot be deleted. Otherwise, no anomaly can be detected when $X$ is written in $b_9$, since this write does not conflict with the read in $b_3$.

More generally, a block $b$ in the reader set of a variable $X$ is subtracted from the access history of $X$ when a block that is a descendent of $b$ accesses $X$. Hence, events performed by blocks which are "vertically" related in the partial order execution graph are combined. An access history contains two blocks $b_i$ and $b_j$ only if $b_i$ and $b_j$ are concurrent and therefore may store at most $T$ readers. On the other hand, since two concurrent writes always conflict, there is at most one writer in an access history.

When block $b$ reads $X$, a test is performed to determine whether $b$ is concurrent with the writer in the access history for $X$ (if one exists). The access history is then updated as follows:

$$Readers(X) \leftarrow Readers(X) \cup \{b\} \text{ - } Ancestors(b)$$

When block $b$ writes $X$, tests are performed to determine whether $b$ is concurrent with any of the events in the access history of $X$, and the access history is updated as follows:

$$Writers(X) \leftarrow b$$
$$Readers(X) \leftarrow Readers(X) \text{ - } Ancestors(b)$$

The cost per block for maintaining access histories is bounded in the worst case by $O(TV)$, which makes this method appear worse than merging. However, we show in Section 6 that this bound is much too pessimistic. Moreover, maintaining access histories rather than Read and Write sets provides better diagnostic capability. Since anomalies are detected as soon as they occur, it is possible to identify the precise location of the anomaly.

The method used to test for concurrency is the primary difference between the English-Hebrew labeling and the task recycling algorithms which are presented below.

### 3.2.1 English-Hebrew Labeling

In the English-Hebrew labeling algorithm, the structure of the partial order execution graph is encoded in *tags* [13,12] associated with blocks. One can determine whether two blocks are concurrent simply by comparing their tags. A tag consists of a pair of labels: an *English label* $E$ and a *Hebrew label* $H$. Conceptually, the English label is produced by performing a left-to-right preorder numbering of the partial order execution graph; each block is assigned a number greater than all of the numbers assigned to its children and its siblings to the right. However, since this label must be generated on-line, a complete traversal of the execution graph cannot be performed. Therefore, a label is a string of numbers and labels are *lexicographically* ordered.

The English label of the root block of a program execution graph is 1. The children blocks $b_{c_0} \ldots b_{c_f}$ of a fork vertex with parent block $b_f$ are assigned English labels:

$$\text{fork:} \quad E(b_{c_i}) \leftarrow E(b_f) \mid i$$

where | is the append operation. The child block $b_j$ of a join vertex with parents $b_{p_0} \ldots b_{p_f}$ is assigned English label:

$$\text{join:} \quad E(b_j) \leftarrow max(E(b_{p_i}))$$

The Hebrew labels are created symmetricly for a right-to-left ordering; the children of fork and join vertices are labeled as follows:

$$\text{fork:} \quad H(b_{c_i}) \leftarrow H(b_f) \mid (f + 1 - i)$$
$$\text{join:} \quad H(b_j) \leftarrow max(H(b_{p_i}))$$

As specified above, the length of the labels increase with the number of fork and join operations. However, an additional heuristic described in [13] bounds the label length by $O(N)$. Figure 5 shows the English-Hebrew labels for a partial order execution graph.

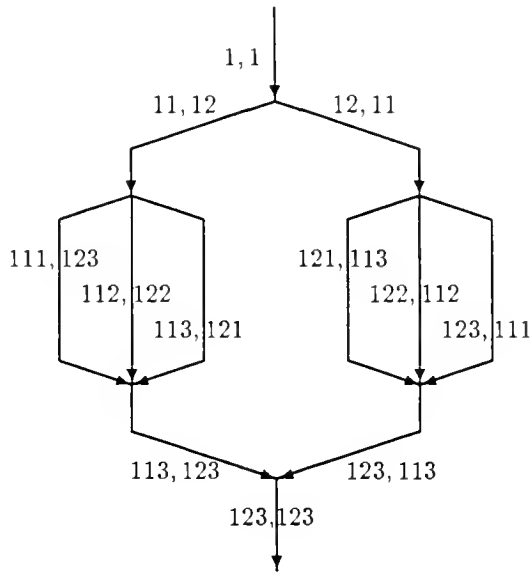



Figure 5: English-Hebrew Labeling of a Partial Order Execution Graph

Two blocks $b_i$ and $b_j$ are concurrent iff either of the following conditions is met :

$$E(b_i) < E(b_j) \text{ and } H(b_i) > H(b_j) \text{ or}$$
$$E(b_i) > E(b_j) \text{ and } H(b_i) < H(b_j)$$

If neither condition is met the tags are said to be *ordered.* Otherwise the tags are said to be *unordered.* For example, in Figure 5, tags $11,12$ and $123,111$ are associated with two concurrent blocks and are unordered, since the first condition is satisfied: namely, that $11 < 123$ and $12 > 111$. However, the two blocks with tags $11,12$ and $113,121$, are not concurrent and the tags are ordered (specifically, $11,12$ is an ancestor of $113,121$).

Unlike the merge algorithm, the partial order execution graph is explicitly encoded in tags so that the storage cost for concurrency information must be considered. Access histories may store tags either directly or indirectly. If the tag itself is stored, the size of each access history is $O(NR)$. There is also a larger constant overhead because tags are variable length. On the other hand, if tags are stored indirectly (i.e. if access histories consist of pointers to tags), tags must be saved throughout the execution requiring storage proportional to the number of blocks.

### 3.2.2 Task Recycling Algorithm

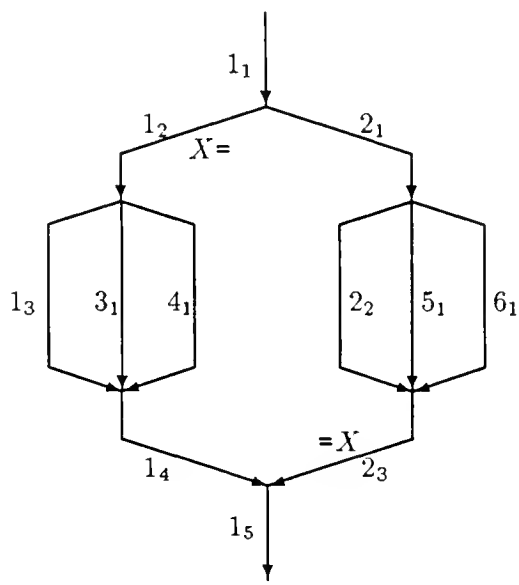
Task Recycling also uses access histories but encodes concurrency information in a different manner which reduces the storage requirements, as well as the cost of testing whether two blocks are concurrent, by a factor of $O(N)$. Instead of a tag, each block has a unique *task identifier*, which consists of a *task* and a *version number.* Tasks may be recycled; that is, more than one block may be assigned to the same task at different times in the program execution. A block $b$ is *validly* assigned to a task $t$ iff it is not concurrent with any other block previously

assigned to $t$. The goal of task assignment is to minimize the number of tasks used while maintaining a valid task assignment. The lowest achievable bound is given by the following theorem:

**Theorem 2:** The minimum number of tasks needed to perform a valid assignment to a partial order execution graph is equal to the maximum concurrency of the graph.

Any task assignment that uses this minimum number of tasks is said to be *optimal.* The version number of a task identifier is used to distinguish among different blocks assigned to the same task. Every time a block is assigned to a task $t$, the associated version number $v$ is incremented.

Unlike English-Hebrew labels, it is not possible to determine whether two blocks are concurrent by simply examining task identifiers. Therefore, additional concurrency information is maintained in a *parent vector* which is associated with each block[2]. Each vector is of length $T$; the $t^{th}$ entry in the parent vector for block $b$ contains the largest version number associated with those ancestors of $b$ which were assigned to task $t$. Therefore, a block $b$ is concurrent with a block with task identifier $t_v$ iff $parent_b[t] < v$. Figure 6 shows an optimal task assignment and parent vectors for a partial order execution graph.




| *Task Ids* | *Parent Vector* |
|---|---|
| $1_1$ | [0,0,0,0,0,0] |
| $1_2, 2_1$ | [1,0,0,0,0,0] |
| $1_3, 3_1, 4_1$ | [2,0,0,0,0,0] |
| $2_2, 5_1, 6_1$ | [1,1,0,0,0,0] |
| $1_4$ | [3,0,1,1,0,0] |
| $2_3$ | [1,2,0,0,1,1] |
| $1_5$ | [4,3,1,1,1,1] |

Figure 6: Task Recycling Assignment of a Partial Order Execution Graph

New parent vectors are formed from those vectors associated with parent blocks. If a block $b_p$ with task identifier $t_v$ performs a fork operation creating child blocks $b_{c_1}$ ... $b_{c_f}$, the parent vectors of $b_{c_1}$ ... $b_{c_f}$ are identical to that of $b_p$ except that the $t^{th}$ entry takes the value $v$. If blocks $b_{p_1} \ldots b_{p_f}$ with task identifiers $t_{1v_1} \ldots t_{fv_f}$ perform a join creating a child block $b_c$, the parent vector of $b_c$ is initialized as follows:

[2]Parent vectors correspond to *before* vectors in [6,11]. However, because we monitor on-line, we do not need the corresponding *after* vectors used for post-mortum trace analysis.

```
for i = 1 to T do
    if ∃_{t_j ∈ {t_1 ... t_f}} : i = t_j then
        parent_{b_c}[i] ← v_j
    else parent_{b_c}[i] ← max(parent_{b_{p_1}}[i], ..., parent_{b_{p_f}}[i])
endfor
```

Unlike English-Hebrew labels, *once a block terminates, the parent vector for the block is no longer needed*, so that the storage for parent vectors is independent of the number of blocks.

To illustrate anomaly detection consider the task assignment in Figure 6. Suppose $X$ is written in block $1_2$ and subsequently read in block $2_3$. After the write, the write access history of $X$ contains $1_2$. When the read occurs, the parent vector of $2_3$ is compared with the task identifier stored in the access history of $X$. The version number for task 1 in the parent vector of block $2_3$ is 1 which is less than the version number 2 stored in the access history; therefore an anomaly is indicated.

The task assignment algorithm - from which this method derives its name - is based on a "most recently used" heuristic. Consider the partial execution graph in Figure 7a. Blocks $b_1$

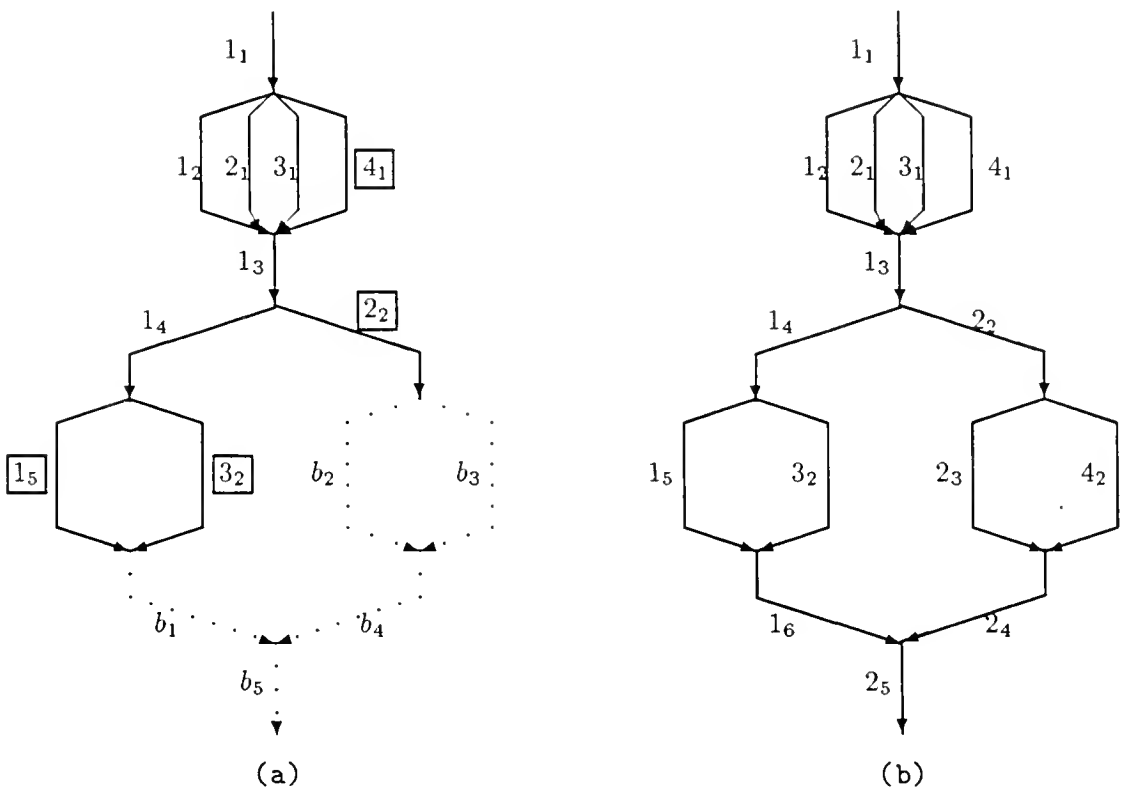



Figure 7: Optimal Task Assignment to Partial Order Execution Graph.

$\ldots b_5$ are unassigned and tasks 1, 2, 3, and 4 are available for reassignment. Suppose $b_1$ is the next block to be assigned. Block $b_1$ may be validly assigned either task 1, 3, or 4; it cannot be assigned to task 2 because it is concurrent block $2_2$. However, if task 4 is used the overall assignment cannot be optimal: a new task will have to be created to assign to either $b_2$ or $b_3$. Figure 7b shows an optimal assignment made by the MRU task assignment algorithm.

To implement the MRU algorithm, a *free task dag* is maintained. There is a vertex in the dag associated with every fork operation, every join operation, and every currently executing block. (These latter vertices are always leaf vertices). The direction of edges in the dag is the reverse of the execution flow. A set of free tasks, i.e. those tasks that are available for recycling, is associated with each vertex. When a block $b$ terminates at a fork or join operation $O$, it adds its task to the free task list of the vertex $v_O$ associated with $O$. The vertex $v_b$ associated with the block is then deleted, and new edges are created from $v_O$ to the children of $v_b$. When a block $b$ is created after a fork or join operation $O$, an edge is added from the vertex $v_b$ associated with $b$ to vertex $v_O$. To obtain a free task, a topological traversal of the dag is performed starting at $v_b$ until a vertex with a non-empty free task set is found. Two optimizations minimize the cost of this topological traversal. Whenever a free task set of a vertex $v$ becomes empty, the dag is *collapsed* by deleting $v$ from the dag and adding edges from the parent vertices of $v$ to children vertices of $v$. Whenever a vertex $v$ has a single parent vertex $p$, $v$ is *subsumed* by $p$ by adding the free task set of $v$ to the free task set of $p$, deleting $v$ and adjusting the edges as appropriate. Figure 8 shows the state of the dag at specific stages in the execution of the example program in Figure 7.

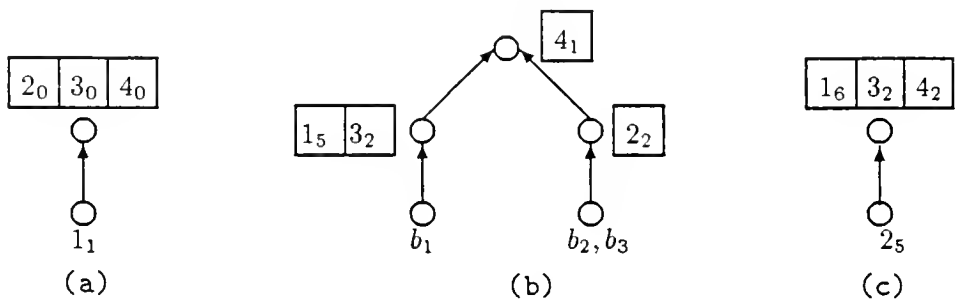



Figure 8: Stages in Free Task Dag for Partial Order Execution Graph in Figure 7.
Figure 8a shows the dag before the first fork operation.
Figure 8b shows the dag when the program is in the state shown in Figure 7a.
Figure 8c shows the dag when block $2_5$ is executing

The following properties hold for the MRU algorithm[3]:

**Theorem 3:** The task assignment performed by the MRU algorithm is always optimal on series-parallel partial order execution graphs.

**Theorem 4:** The size of the free task dag used by the MRU algorithm on series-parallel partial order execution graphs is bounded by $O(T)$.

**Theorem 5:** The amortized cost of assigning a task to a block for the MRU algorithm on series-parallel partial order execution graphs is $O(N)$. [4]

There is a constant cost (an array access) for checking whether the current block is concurrent with an access history entry, so that the overall cost per variable access is $O(R)$. Since task

[3]These properties hold because the free task dag is actually a tree with height $O(N)$. As will be seen in Section 5, these properties no longer hold for general partial order execution graphs.

[4]If the size of each fork-join construct is greater than $2N$, the number of vertices in the dag is $O(\frac{P}{N})$ and the amortized work is O(1).

identifiers, which consist of two integers, are stored directly in the access history, the size of an access history is also $O(R)$. By Theorem 3, the length of each parent vector is $O(T)$. The number of parent vectors at any one time is bounded by the number of blocks that can run concurrently, so that the total space for parent vectors is $O(T^2)$.

### 3.3 Conclusion

The average and worst case costs for both performance and storage are presented in Table 2. We first compare access history based methods with merging, and then compare the two

| | | *Time (Per Block)* | | *Space (Overall)* | |
|---|---|---|---|---|---|
| | *Algorithm* | *Creation & Termination* | *Variable Access* | *Concurrency Information* | *Read-Write Set* |
| Average Case | Merge | $O(V)$ | $O(A)$ | 0 | $O(VT)$ |
| | E-H Label | $O(N)$ | $O(ANR)$ | $O((T+VR)N)$ or $O(BN)$ † | $O(VR)$ |
| | Recycling | $O(T)$ | $O(AR)$ | $O(T^2)$ | $O(VR)$ |
| Worst Case | Merge | $O(V)$ | $O(V)$ | 0 | $O(VT)$ |
| | E-H Label | $O(N)$ | $O(VNT)$ | $O(VTN)$ or $O(BN)$ † | $O(VT)$ |
| | Recycling | $O(T)$ | $O(VT)$ | $O(T^2)$ | $O(VT)$ |

† The first number is for storing tags directly in access histories; the second for using indirection

Table 2: Average and Worst Case Comparison of Algorithms

access history algorithms. To maintain access histories, the number of events that must be examined per block is $O(VT)$. However, on average the cost per block is a function of $A$, the number of variables actually accessed in each block. Furthermore, as is shown in Section 6, the average number of readers in an access history $R$ is much less than $T$. In fact, empirical results indicate that $R$ is often independent of $T$. This property results from the common programming practice of data partitioning, whereby different instruction threads access disjoint sets of memory addresses in order to optimize performance. Therefore, the actual cost per block $O(AR)$ for access history algorithms is significantly less than the worst case bound of $O(VT)$.

The actual cost of the merge algorithm depends somewhat on the data structures used. If a bitmap representation is used, the cost per variable access is small, but the average case storage cost is the same as the worst case, as seen in Table 2. Alternatively, a list-based representation of Read and Write sets may be implemented, which uses storage more efficiently when Read and Write sets are sparse. However, the cost per variable access is thereby increased, and the sizes of variable-length Read and Write sets increase after each merge operation. Because there are so few concurrent readers, the size of the merged Read Set tends to grow proportionally to the number of merged sets. Therefore, the worst case $O(V)$ for merging is a tighter bound than the bound for access histories. It follows that the merge algorithm may perform better than access history algorithms in worst case scenarios: namely, when the number of variables accessed per block $A$ is large. Because no explicit concurrency information is required, merging should also perform well for programs with fine-grained and highly dynamic parallelism.

Of the two event history algorithms, task recycling is more efficient than English-Hebrew labeling in two ways: the cost per variable access is less, and the space needed for concurrency information is smaller. However, because fork and join operations are cheaper, English-Hebrew labeling may have a performance advantage when parallelism is fine-grained, when fork and join operations are frequent, when $T$ is large, and/or when there is no nested fork and join constructs.

Finally, task recycling and the merge algorithms benefit from a "run-until-completed" scheduling paradigm, which is the common scheduling method used in parallel Fortran environments. Parallel scientific codes typically exhibit a high degree of *nominal* parallelism. In the "run-until-completed" model, each block created in a fork will not block until it terminates at its associated join operation. This limits the number of blocks which can ever run concurrently to $n$, the underlying parallelism of the machine. For the task recycling algorithm, this means that there can be at most $n$ parent vectors at any point in time. For the merge algorithm, the number of Read-Write sets for currently executing blocks is also bounded by $n$. Therefore, the space requirements for these data structure can be reduced by a factor of $\frac{n}{T}$. No similar space savings hold for the English-Hebrew labeling scheme. Most of its space requirement is for saving a history of block labels.

These issues are further discussed in Section 6. Sections 4 and 5 introduce general inter-thread coordination and describe the implications for the three algorithms. For programs with frequent coordination, the task recycling algorithm appears considerably better than the other two algorithms.

# 4 General Concurrency Model

Series-parallel partial order execution graphs model only fork-join parallelism and hence covers a limited class of parallel programs. We now extend the basic model to include more general thread coordination[5] and thereby encompass virtually all parallel programs. Coordination is either *synchronous* or *asynchronous*. If two threads coordinate synchronously (e.g. via a barrier operation) neither thread may execute beyond the coordination point until both have reached it. If two threads coordinate asynchronously (e.g. via a signal operation) the receiver may not execute beyond the coordination point until the sender has reached it; however, the sender may proceed immediately.

The partial order execution graph definition is extended as follows. A block is an instruction sequence executed by a single thread that does not include fork, join, or coordination operations. An edge is either a block edge or a *coordination edge*, which connects vertices of two coordinating threads. A vertex may be one of three additional types:

- A *synchronous vertex* has an associated synchronous vertex, two out edges (the child block and a coordination edge to the associated synchronous vertex), and two in edges (the parent block and a coordination edge from the associated synchronous vertex).
- A *sender vertex* has an associated receiver vertex, two out edges (the child block and a coordination edge to the receiver vertex), and one in edge (the parent block).
- A *receiver vertex* has an associated sender vertex, one out edge (the child block), and two in edges (the parent block and a coordination edge from the sender vertex).

Coordination introduces additional ordering among blocks not captured by considering only fork-joins. The maximum concurrency for a graph with coordination edges may be less than the maximum concurrency for the same graph with coordination edges removed. (We denote the maximum concurrency of a partial order execution graph with coordination edges removed as $T'$.) Figure 9 illustrates a partial order execution graph with a synchronous coordination operation between blocks $b_2$ and $b_9$. In Figure 9, the maximum concurrency is 4; the maximum

[5]Coordination primitives include, for example, barrier synchronization, events, Ada rendezvous, message passing, and locks.

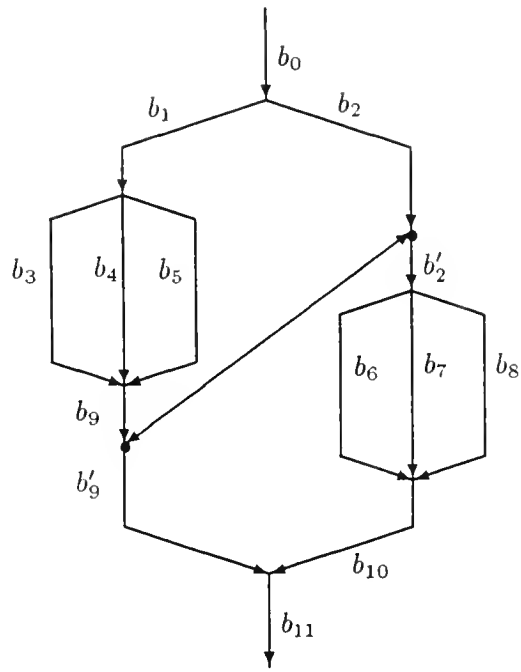



Figure 9: Partial Order Execution Graph with Two Blocks Synchronizing

concurrency is 6 if the coordination edge between $b_2$ and $b_9$ is removed. An *indirect parent* (resp. *indirect ancestor*) is a block that is a parent (resp. ancestor) because of an explicit coordination edge. In Figure 9, $b_2$ is the *direct parent* of $b'_2$, and $b_9$ is the *indirect parent* of $b'_2$. Similarly, $b_9$ and $b_2$ are (direct and indirect) parents of block $b'_9$.

Coordination also increases the difficulty of anomaly detection. Asynchronous coordination, in particular, introduces the following problems:

- Partial order execution graphs with asynchronous coordination may have cycles of arbitrary length. However, we may still assume that execution graphs are acyclic, as a result of the following:

  **Theorem 6:** There is a cycle of length greater than two in a general partial order execution graph iff the blocks in the cycle are deadlocked.

  Thus anomalies in cyclic graphs are hidden by the larger problem of deadlock. On the other hand, synchronous coordination creates degenerate cycles of length two, which are special-cased by the algorithms.

- When two or more blocks $b_i \ldots b_j$ coordinate, the detection methods require that the concurrency information of $b_i \ldots b_j$ be available when the coordination occurs. For asynchronous coordination, this means that whenever a sender block reaches a coordination point before the receiver, its concurrency information must be saved until the receiver reaches the coordination point. Therefore, there is an additional cost for buffering concurrency information in all three algorithms. [6]

[6]The concurrency list and Read and Write sets must be saved for the merge algorithm, the coordination list must be saved for English-Hebrew labeling, and the parent vector must be saved for task recycling.

Intuitively, since increasing coordination reduces concurrency, access histories should be smaller and anomaly detection more efficient for this more general model of execution. However, partial order relationships are more difficult to determine for graphs that are not series-parallel. This is reflected in increased computational and space requirements for concurrency information. The algorithm extensions necessary to handle coordination edges is the topic of the next section.

# 5 Algorithm Extensions For Coordination

The algorithms presented in Section 3 require extensions to deal with both synchronous and asynchronous coordination. The examples used in this section illustrate synchronous coordination; the special problems involving asynchronous coordination are described above and not further addressed. In all three algorithms, additional processing at coordination points is required. However, since both basic merging and English-Hebrew labeling rely heavily on properties of series-parallel graphs, these algorithms require completely new mechanisms to determine concurrency relationships among blocks are needed. This is not the case for task recycling, which extends in a straightforward manner.

## 5.1 Merge Algorithm

In order to determine concurrency relationships in graphs with coordination edges, *concurrency lists* are used [14]. A concurrency list is associated with one or more blocks, and Read and Write sets are associated with concurrency lists. A concurrency list associated with a block $b$ is the set of all tasks $T$ such that the block $b'$ currently executing in $T$ is concurrent with $b$. When a block $b'$ terminates, its Read and Write set is compared with all Read and Write sets associated with concurrency lists containing $T$. Read and Write sets are merged whenever their associated concurrency lists are equal. More precisely, when a block $b'$ in task $T$ completes, the following steps are performed:

1. Compare the Read and Write set for $b'$ with all Read and Write sets whose concurrency list contains $T$.
2. Remove $T$ from all concurrency lists associated with blocks that are not concurrent with $T$ after $b'$ completes.
3. Merge Read and Write sets with equal concurrency lists.
4. Delete any Read and Write sets associated with empty concurrency lists.

The maximum number of concurrency lists is $O(T^2)$ so that $O(T^3)$ space is needed to store concurrency lists. (See [14] for a more complete description). Since there is one Read and Write set pair per concurrency list, the storage for Read-Write set pairs is $O(T^2V)$, and the performance cost at each fork, join, or coordination is $O(T^2V)$. Thus, there is an overall performance degradation by a factor of $T^2$ in the worst case compared to the original merge algorithm presented in Section 3.1.

## 5.2 English-Hebrew Labeling

English-Hebrew tags only encode concurrency properties for series-parallel graphs. An additional mechanism is needed to record execution orderings imposed by more general forms of coordination. To this end, a *coordination list* is associated with each block $b$ which contains

the tags of all blocks that coordinated with $b$ or with a parent of $b$. Coordination lists are therefore similar in function to parent vectors. However, in contrast to parent vectors, coordination lists need to store only indirect parents. Also coordination lists may not be stored like parent vectors as direct-access arrays, since the domain (all tags) of a coordination list is sparse. Linear search structures must be used instead. Like parent vectors, coordination lists are maintained only for blocks currently executing; therefore, the number of coordination lists is bounded by $T$.

The coordination list of a new block $b$ consists of the union of the tags of the indirect parents of $b$ and the coordination lists of all direct and indirect parents of $b$, such that all tags in the coordination list are unordered. (See Section 3.2.1 for definition of unordered.) Thus, if block $b_i$ is a direct ancestor of another block $b_j$, then the tags labeling $b_i$ and $b_j$ can never be in the same coordination list. The length of each coordination list is therefore bounded by $T$, so that $O(T^2)$ additional space is required. More precisely, when a block $b_c$ is created with parent blocks $b_{p_1} \ldots b_{p_m}$, the coordination list $list_{b_c}$ is created as follows:

for every indirect parent $b_{p_i}$ add $tag_{b_{p_i}}$ to $list_{b_c}$
for every parent $b_{p_i}$ and every $tag_{b_j}$ in $list_{b_{p_i}}$
    if $unordered(tag_{b_j}, tag_{b_c})$ and $\forall_{tag_{b_k} \text{ in } list_{b_c}}\ unordered(tag_{b_j}, tag_{b_k})$ then
        add $tag_{b_j}$ to $list_{b_c}$

Figure 10 shows the tags and coordination lists both before and after synchronization for a portion of a partial order execution graph. The coordination list for block 21,71 in Figure

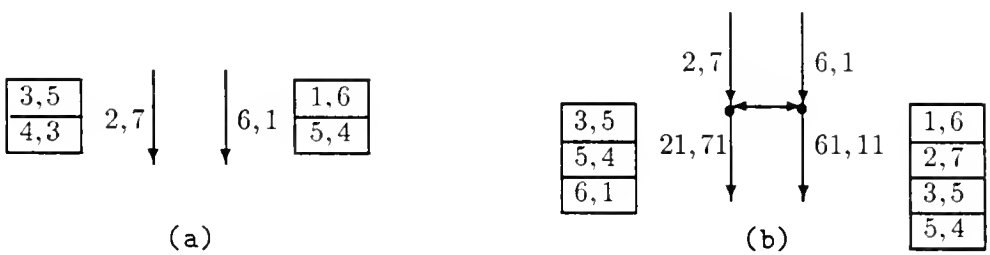



Figure 10: Tags and Coordination Lists for the Modified Program Execution Graph

10.b contains the indirect parent tag 6,1. Tags 5,4 and 3,5 are added from the coordinations lists of the parents of 21,71. All of these tags are unordered. However, 1,6 is not added to the coordination list because it is ordered with respect to 21,71, and 4,3 is not added because it is ordered with respect to 5,4. The coordination list for block 61,11 is formed in a similar manner.

To test for concurrency between two blocks, coordination lists are used to gain information about indirect ancestors. Suppose block $b_i$ accesses variable $X$, which has $tag_{b_j}$ in its access history. If $tag_{b_i}$ and $tag_{b_j}$ are ordered, then block $b_j$ is a direct ancestor of block $b_i$. Otherwise, $tag_{b_j}$ must be compared with each tag stored in the coordination list associated with $b_i$. If any tag in the list and $tag_{b_j}$ are ordered then $b_j$ is an indirect ancestor of $b_i$. Otherwise, $b_j$ and $b_i$ are concurrent. Hence, when a block $b$ accesses variable $X$, every entry in the coordination list of $b$ may have to be compared to every tag in the access history of $X$. In a naive implementation, the worst case cost per variable access is increased to $O(NT^2)$. However, if tags are maintained in sorted order, this cost is reduced. On average, the cost per variable access is bounded by $O(N(R+C))$, where $C$ is the average length of a coordination list.

## 5.3 Task Recycling

Unlike the two previous algorithms, task recycling is extended in a straightforward manner. When blocks $b_i$ and $b_j$ coordinate, new task identifiers are formed by incrementing the version numbers of the task identifiers associated with $b_i$ and $b_j$: new parent vectors are created by applying the rules described in Section 3.2.2. The only complication is in extending the task assignment algorithm, as described below.

First we consider the properties of task assignments for more general partial order graphs. The MRU assignment algorithm in Section 3.2.2 obtains an optimal task assignment for graphs without coordination edges. However the maximum concurrency of a graph $T$ with coordination edges may be less than the maximum concurrency without the coordination edges $T'$. In fact, we show in the Appendix that Theorem 2 still holds for general partial order execution graphs, namely, that the smallest task assignment is equal to $T$. The following theorem gives an upper bound for computing the optimal assignment.

**Theorem 7:** There exists an offline optimal task assignment algorithm for general partial order execution graphs which requires $O(B^{2.5})$ work.

For example, Figure 11 shows an optimal task assignment for the graph in Figure 9, where the size of the assignment is equal to the maximum concurrency of 4.

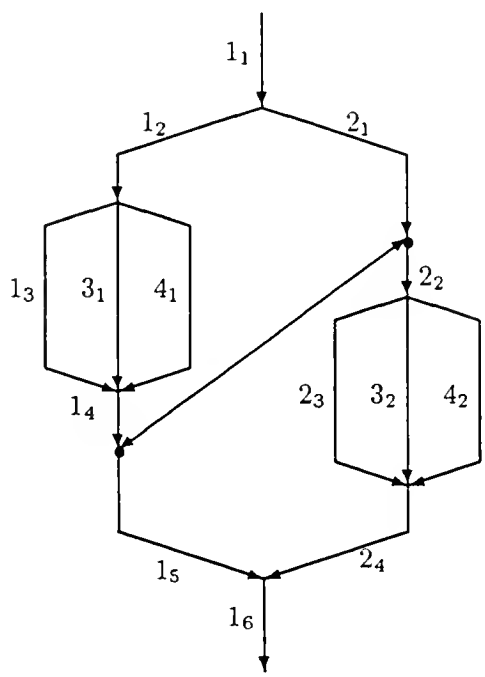



Figure 11: Partial Order Execution Graph with Two Blocks Synchronizing

Unfortunately, however, it is not possible to obtain the optimal task assignment using the MRU or any other online algorithm:

**Theorem 8:** A worst case lower bound on the number of tasks needed for any online task assignment algorithm on general partial order execution graphs is $\frac{3T}{2} - log(T)$.

Nevertheless we can extend the MRU algorithm to handle coordinations, with the result that, while not guaranteed to be optimal, the algorithm works well in practice. When a block $b$ terminates in a coordination operation, a new vertex $v_O$ is created with edges to the vertices associated with all coordinating blocks and the task of $b$ is added to this new vertex. To assign a task to a block $b$, a topological traversal of the dag is performed starting from the block associated with the block $b$.

To illustrate the MRU algorithm extension, Figure 12 shows three stages in a partial order execution graph containing a synchronous coordination operation. Figure 12.a shows four

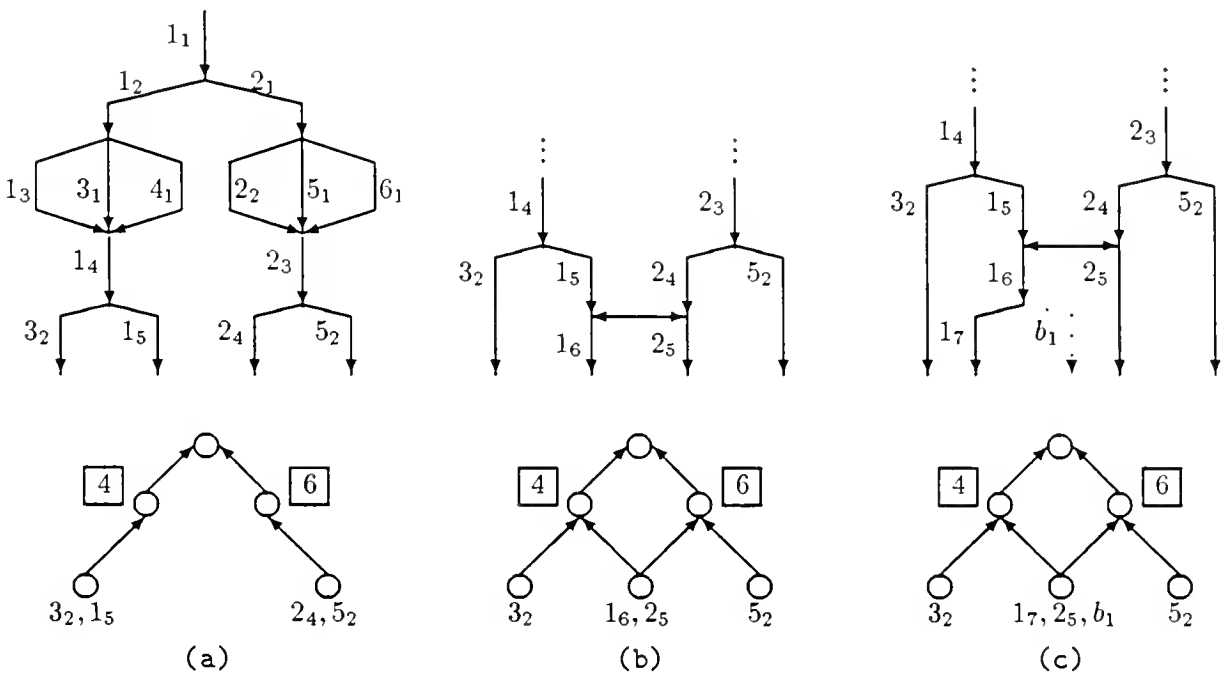



Figure 12: Task Labels and Free Task Dag for Three Phases of a Partial Program Execution Graph

blocks executing, with task identifiers $3_2$, $1_5$, $2_4$ and $5_2$. In the associated free task dag, shown below the execution graph, task 4 is reachable from blocks $3_2$ and $1_5$, and task 6 is reachable from $2_4$ and $5_2$. Figure 12.b shows the changes made to the partial order execution graph and free task dag after a synchronous coordination operation between blocks $1_5$ and $2_4$. Free tasks 4 and 6 are both now reachable from blocks $1_6$ and $2_5$. Consequently more choices are available to assign to children of these blocks. Figure 12.c shows a third phase in the execution of the program in which block $b_1$ has been created but is not yet assigned. There are two choices for $b_1$, namely either task 4 or 6. However, depending on future operations, the choice made may result in an assignment that is not optimal. Suppose $b_1$ uses task 4. If, at some time in the future, block $3_2$ creates two new threads, a new task must be generated. This forces seven tasks to be used in the assignment even though the graph has a maximum concurrency of six. A symmetric scenario holds if $b_1$ uses task 6. Thus, it is not possible to guarantee an optimal assignment because the future operations performed by blocks $3_2$ and $5_2$ are unknown. Intuitively, the additional complexity of optimal task assignment stems from the fact that the

free task dag is no longer a tree and there is a choice as to which vertex to use tasks from.

The extended MRU algorithm has the following properties:

**Theorem 9:** The size of the free task dag used by the MRU algorithm on general partial order execution graphs is bounded by $O(T^2)$. [7]

**Theorem 10:** The amortized cost of assigning a task to a block for the MRU algorithm on general partial order execution graphs is $O(T)$.

Note that the MRU algorithm will never assign more than $T'$ tasks at worst.

## 5.4 Comparison

Table 3 summarizes the worst case space and time requirements for the extended algorithms. (Average case requirements can no longer be validly parameterized because of the complexity of measuring costs stemming from coordinations.) We compare the relative cost increases

| | *Time (Per Block)* | | *Space (Overall)* | |
|---|---|---|---|---|
| *Algorithm* | *Creation & Termination* | *Variable Access* | *Concurrency Information* | *Read-Write Set* |
| Merge | $O(T^2V)$ | $O(V)$ | $O(T^3)$ | $O(VT^2)$ |
| E-H Label | $O(NT)$ | $O(VTN)$ | $O((T+V)TN)$ or $O(T^2+BN)$† | $O(VT)$ |
| Recycling | $O(T')$ | $O(VT)$ | $O(TT')$ | $O(VT)$ |

† The first number is for storing tags directly in access histories; the second for using indirection

Table 3: Worse Case Comparison of Algorithms

for the three algorithms to handle graphs with coordination edges. The merge algorithm now must maintain concurrency information explicitly, and the overall worst case cost increases by a factor of $T^2$. While the average cost is much less than this worst case [14], there is nevertheless a significant degradation in space and performance compared to the basic merge algorithm. The additional overhead for English-Hebrew labeling is the maintenance of coordination lists at block termination, and the cost of checking coordination lists at each variable access. If coordination lists are long because of frequent coordinations, this additional cost can degrade the performance significantly. Task recycling is the least affected. The cost of maintaining the free task dag is slightly more expensive, but this represents a small additional overhead at block termination. We therefore conclude that when coordination is considered, task recycling appears to have an even greater advantage over the other algorithms than it did before.

Several generalizations can be made about the class of programs each algorithm will monitor most efficiently. Unless concurrency and coordination is very limited (few Read and Write sets are needed), or the degree of parallelism is high and there is no coordination (very large parent vectors are needed), the merge algorithm is much less efficient than the access history based algorithms. If there is a high degree of parallelism with limited coordination, English-Hebrew labeling may also perform better than task recycling because of the high cost of maintaining parent vectors and task assignment. However, there is almost always a large storage penalty for saving tags in the English-Hebrew labeling algorithm. Moreover, if there is a large amount of coordination, the cost of maintaining coordination lists approaches the cost

[7] The theoretical upper bound of $O(T^2)$ is required only for certain pathological partial order execution graphs. In general, the size of the free task dag, and hence the amortized cost for task assignment, will be much less.

of maintaining parent vectors, so that any advantage of English-Hebrew labeling is diminished. These observations are substantiated in Section 6.

When subtraction is combined with merging, the theoretically minimum number read and write events is obtained [14]. In order to combine these two techniques, list based (rather than bitmap) representations of Read and Write sets must be used. The worst case costs for the merge algorithm thereby decreases by a factor of $T$. The space required for concurrency information remains the same. The number of read and write events is minimized because there are space reductions that are possible from merging but not subtraction, and space reductions possible from subtraction but not from merging. However, because $R$ is in practice quite small (see Section 6), the marginal savings from merging is also quite small. Additionally, there is a much higher overhead for each variable access and at synchronization points. Perhaps heuristics which decrease the space by guaranteeing that only a high percentage of variables which are accessed in an "unsafe" manner are detected are a more fruitful method of optimization.

# 6 Empirical Results

The preceding sections present algorithms for detecting access anomalies in parallel programs. Discussions of algorithm design and analysis, however, give incomplete insight into the costs encountered when detecting anomalies in actual parallel programs. To this end we present measurements from several benchmark scientific parallel programs:

| | | |
|---|---|---|
| *Triso* | - | Solves a sparse triangular linear system of equations using "wavefront" parallelism; it takes as input a directed acyclic graph and processes each level of the graph in parallel. |
| *Finite* | - | Solves a linear system using finite element methods |
| *Simple* | - | Solves partial differential equations for hydrodynamics and heat conduction. |
| *Polymer* | - | Performs molecular dynamic calculations of polymer systems. |

Our first goal using these benchmark programs is to obtain representative values for important parameters that measure concurrency structure ($T$ and $B$) and shared variable access patterns ($V$, $A$, and $R$). From these values, it is possible to better compare the performance costs specified in Table 3. Our second goal is to measure the actual performance impact of monitoring parallel programs. These measurements were obtained by implementing several of the detection algorithms on the NYU Ultracomputer [8].

## 6.1 Concurrency Structure Parameter Measurements

For all four benchmark parallel programs, the concurrency structure is quite simple: there is very limited nesting of fork-join constructs and minimal synchronization. However the degree and granularity of parallelism vary considerably.

- Triso has coarse granularity parallelism with limited synchronization. It consists of a single fork-join operation which creates 8 parallel threads; these subsequently perform two *barrier* synchronization operations (represented by an 8-way synchronous coordination in the partial order execution graph).
- Simple has medium granularity parallelism with some locking. It performs 10 fork-join operations that each create 124 parallel threads, and 130 operations that create from 10 to 30 threads. In addition, during 10 phases of execution approximately half of

the 30 concurrent threads obtain a lock (represented by an asynchronous coordination operation).

- Finite exhibits a large degree of fine granularity of parallelism and uses the pure fork-join model. It performs 60 fork-join operations that each create 1000 parallel threads; 50 operations that create 250 threads and 200 operations that create between 2 and 32 threads. Each block performs a very limited amount of computation; in many case a block consists of a single operation on one element of an array.
- Polymer also exhibits a large degree of fine granularity parallelism and uses the pure fork-join model; however, it has one level of nested fork-join operations (the first three benchmark programs have no nested fork-join operations). It performs 40 nested fork-join operations; the outer operations create 1000 parallel threads each of which creates 3 parallel sub-threads. In addition, it performs 20 fork-join operations which create 350 parallel threads and 10 fork-join operations which create 100 parallel threads.

Table 4 summaries the concurrency parameters $B$, $T$, and the average value of $T$ for the benchmark programs. The experimental results presented in Section 6.3 show that the concurrency

| *Program* | $B$ | $T$ | $T_{ave}$ |
|---|---|---|---|
| Triso | 24 | 8 | 8 |
| Simple | 4090 | 124 | 28 |
| Finite | 74,900 | 1,000 | 245 |
| Polymer | 128,000 | 3,000 | 1,800 |

Table 4: Concurrency Structure

structure of the program has a significant impact on the cost of maintaining concurrency information. However, the results also show that the concurrency structure does not significantly impact the cost per variable access, as discussed in the next section.

## 6.2 Shared Variable Access Parameter Measurements

The average size of the reader set of an access history $R$ determines in part the benefit of access history based algorithms over the merge algorithm, since the work and space required for maintaining access histories is proportional to $R$. While theoretically $R$ may grow to $T$, in practice the number of concurrent readers is much more limited since many parallel scientific codes distribute workloads by partitioning data among concurrent threads. Hence a thread often shares data with a neighboring thread, but seldom shares data with all other threads. Therefore, one would expect the number of concurrent readers to be limited.

The shared memory access patterns measured for the benchmark programs support this conclusion and are shown in Table 5. Each column gives the percentage of accessed[8] shared variables with *at most* the specified number of concurrent readers. In the Triso program, for example, 15% of the shared variables have two concurrent readers at some point during execution, but never have more than two. The last column of Table 5 shows the average maximum reader set size for all variables with less than 10 concurrent readers; (it is not possible to measure reader sets larger than 10 due to memory constraints on the Ultracomputer). For the first three benchmark programs, this includes virtually all shared variables and is an upper bound on $R$. Since for the Polymer program 10% of the variables have more than 9 concurrent readers, the figure in the last column only reflects statistics for 90% of its shared variables. If

[8]Because fixed maximum sized arrays are used, many shared variables are never accessed.

| Program | Percentage of Variables with Reader Set Sizes | | | | | | | | | | Ave Size |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | >9 | |
| Triso | 80.2% | 15.0% | 4.8% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 1.25 |
| Simple | 49.3% | 41.0% | 9.6% | † | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | † | 1.69 |
| Finite | 48.4% | 32.5% | 4.6% | † | † | 0.9% | † | † | 13.4% | † | 2.71 |
| Polymer | 72.9% | 9.4% | 2.6% | 1.1% | 2.8% | 0.8% | 2.6% | 6.2% | 2.0% | 9.3% | 1.31 |

† Less than 0.1% of the variables have reader sets of this size

Table 5: Reader Set Sizes

we are conservative and assume that all of these variables are accessed by all 3,000 concurrent block, the average reader set size is 279 concurrent readers (which is still much less than 3,000). If we are slightly less conservative and assume that all of these variables are accessed by $T_{ave}$ concurrent blocks, the average reader set size is 168.

Table 5 reveals three somewhat surprising results:

- The size of reader sets tends to be very small. In all four programs, more than 80% of the variables are never read by more than two concurrent readers and almost 50% are never read concurrently. Moreover, a variable with reader set size of $n$ may actually have a much smaller reader set size throughout most of the execution of the program. Hence our estimate of $R$ is pessimistic with respect to the actual average reader set size.
- There appears to be little correlation between the average reader set size and the degree of parallelism of the program. For example, the Triso and Polymer programs have fairly similar access patterns for the majority of their variables (excepting the 9.3% in the Polymer program with more than 9 concurrent readers). Triso, however, has a low degree of parallelism while Polymer has nested parallelism of a very high degree.
- A direct consequence of the limited reader set sizes is that the estimated number of variables accessed per block $A$, as is shown in Table 6, is much smaller than the total

| Program | $R$ | $T_{ave}$ | $V$ | $A$ |
|---|---|---|---|---|
| Triso | 1.25 | 8 | 12,437 | 2908 |
| Simple | 1.69 | 28 | 17,551 | 1183 |
| Finite | 2.71 | 245 | 12,428 | 159 |
| Polymer | 168 † | 1,800 | 24,431 | 2280 |

†A conservative value for $R$ of 168, rather than 1.31, was used.

Table 6: Estimated Number of Variables Accessed Per Block

number of variables $V$. If a program has $V$ shared variables, average concurrency of $T_{ave}$ and average reader set size of $R$, $A$ is at most $\frac{VR}{T_{ave}}$.

Since $R$ and $A$ are so much smaller than $T$ and $V$ respectively, we conclude that the access history based algorithms will generally be more efficient than the merge algorithm. Relative performance of the two access history based algorithms are further discussed in the next section.

## 6.3 Actual Measurements and Comparison of Algorithms

In order to obtain a more concrete comparison and to measure the overhead of monitoring, we implemented the task recycling and English-Hebrew labeling algorithms with restricted reader sets sizes for the parallel Fortran compiler on the 8 processor NYU Ultracomputer.

(Reader set size restrictions are necessary because of 16 megabyte memory constraints. Because similar restrictions cannot be made for the merge algorithm, it is not implemented.) The parallel Fortran compiler provides the *doall* construct as its primary concurrency primitive. The access anomaly detection is performed by a library package mostly written in C. Routines for maintaining access histories are written in assembler for reasons of efficiency. A front-end pre-processor allocates the storage for access histories and inserts calls to the run-time libraries in the program being monitored. This implementation is relatively portable to other parallel Fortran systems.

Several optimizations were made to task recycling and English-Hebrew algorithms during implementation; these are reflected in the measurements discussed in the following sections.

- Blocks may share a parent vector (or coordination list) if they have identical sets of ancestors. For example, all blocks created by a fork operation share the same parents; one of these blocks must obtain a private parent vector only if it terminates before the join (e.g. by performing a nested fork or coordination operation). For a nested *doall* operation with outer parallelism of $m$ and inner parallelism of $n$, this reduces the number of parent vectors by a factor of $nm$. [9]
- Task recycling benefit from the fact that the Ultracomputer parallel Fortran environment uses a "run-until-completed" scheduling paradigm. Consequently, on an 8 processor Ultracomputer, there are at most 8 parent vectors at any point in time.
- In the Ultracomputer parallel Fortran environment, each of $n$ actual processes perform the work of several parallel blocks. The differences among these blocks' parent vectors are very small. Therefore, in order to reduce the work performed in maintaining parent vectors, a private "template" parent vector is cached for each level of nesting. When a block $b$ updates its parent vector, the process which is performing the work of $b$ also records the modifications. This record is used when block $b$ terminates to update the its children blocks' parent vectors (instead of performing a comparison of all of the entries) and back-out the changes made to the parent vector before executing the next block (instead of reinitializing all of the entries). For pure fork-join programs this reduces the per block cost from $T$ to $N$ (the number of times that a change must be either added or backed out of a parent vector) and the amount of space needed per process is $O(TN)$.
- The size of each access history is restricted because of 16 megabyte memory constraints. While this limitation may result in undetected anomalies, we believe that anomalies will be missed only rarely, since the measurements discussed in Section 6.2 indicate that $R$ is generally very small. Four versions of each program were executed:

  | | | |
  |---|---|---|
  | *Unmon* | - | unmonitored program |
  | *Monitor(1)* | - | monitors every shared variable using a reader set size of one |
  | *Monitor(2)* | - | monitors every shared variable using a reader set size of two |
  | *Concurrency* | - | maintains concurrency information only; it does not monitor any shared variables |

  Each entry in an access history contains a task identifier (or a pointer to the English-Hebrew label) and the line number and a pointer to the function name of the last access; this enables us to identify the locations in the program code where access anomalies occur.

[9] If local memory is available and the cost of accessing shared memory is much higher than local memory, sharing of parent vectors may not be cost-effective.

### 6.3.1 Comparison of Space Requirements

Table 7 compares the memory requirements for each of the four versions of the benchmark programs listed above. The static size is the size of the object module which includes additional

| | *Static Sizes* | | | *Dynamic Sizes* | |
|---|---|---|---|---|---|
| *Program* | *Unmon* | *Monitor (1)* | *Monitor (2)* | *T R* | *E-H* |
| Triso | 163 | 660 | 1,000 | 2 | 8 |
| Simple | 314 | 921 | 1,270 | 6 | 80 |
| Finite | 230 | 745 | 1,020 | 29 | 954 |
| Polymer | 639 | 2,898 | 4,461 | 279 | 7,000 † |

†Estimated Value

Table 7: Actual Space Requirements (in Kbytes)

monitoring code as well as storage needed for access histories. This is virtually the same for both algorithms. The dynamic size is the amount of storage needed for concurrency information, which is allocated at run-time. As is seen in Table 7, English-Hebrew labeling requires substantially more space than task recycling for maintaining concurrency information. In fact, it is not possible to obtain the actual dynamic memory requirements of Polymer for the English-Hebrew labeling scheme; the amount of storage needed for concurrency information exceeds the capacity of the Ultracomputer and hence an estimate was calculated based on the concurrency structure of the partial order execution graph.

### 6.3.2 Comparison of Execution Time Requirements

Table 8 displays user mode execution times for the four benchmark programs using both of the anomaly detection algorithms. English-Hebrew labeling cannot be used to monitor the

| | | *Monitor (1)*† | | *Monitor (2)*† | | *Concurrency* | |
|---|---|---|---|---|---|---|---|
| *Program* | *Unmon* | *T R* | *E-H* | *T R* | *E-H* | *T R* | *E-H* |
| Triso | 3.5 | 19.6 | 20.9 | 22.7 | 24.4 | 4.1 | 4.7 |
| Simple | 202 | 550 | 579 | 598 | 676 | 263 | 237 |
| Finite | 98 | 959 | 704 | 1108 | 1072 | 734 | 407 |
| Polymer | 500 | 4367 | N/A† | 4607 | N/A† | 3330 | N/A† |

†Due to memory constraints, it was not possible to measure this value

Table 8: Actual Execution Times (in seconds)

Polymer program because of memory constraints. For the other three programs, the English-Hebrew labeling algorithm requires more time than task recycling for monitoring variables (as is shown in the run times for *Monitor(1)* and *Monitor(2)*), even when the cost of maintaining concurrency information is less (as in shown in the run times for *Concurrency*). The overall increase in computation time for monitoring, as is seen by comparing *Unmon* and *Monitor(1)*, indicates a 3-fold to 8-fold slowdown for both task recycling and English-Hebrew labeling. Although this cost is high, it is not unreasonable during a debugging phase of program development. Moreover, in general, static analysis or user directives may be used so that only a subset of accesses to shared variables need be monitored. *Concurrency* isolates the cost of maintaining the concurrency information from the overall cost of detecting access anomalies. This cost is substantial for the programs with very high degrees of fine-grained parallelism. In the current implementation of the task assignment algorithm, assigning and freeing a task

requires locking vertices in the free task dag, which creates a serial bottleneck. If free tasks are stored in parallel access data structures (e.g. parallel access queues), it may be possible to decrease this serialization effect.

From Section 6.2 we know that using two entry reader sets instead of one entry reader sets significantly increases the percentage of variables guaranteed to have at least one anomaly detected (from 50% to 80%). A comparison of the execution times for *Monitor(1)* and *Monitor(2)* in Table 8 shows that we can do so at with little additional cost. Table 9 isolates the cost of maintaining access histories and shows the percentage increase when using reader sets of size two instead of one. The isolated times are computed as the total monitored execution time

| | *Monitor (1)* | | *Monitor (2)* | | *Increase* | |
|---|---|---|---|---|---|---|
| *Program* | *T R* | *E-H* | *T R* | *E-H* | *T R* | *E-H* |
| Triso | 16.4 | 16.3 | 21.0 | 22.1 | 20% | 22% |
| Simple | 287 | 242 | 335 | 339 | 17% | 40% |
| Finite | 225 | 297 | 354 | 685 | 57% | 130% |

Table 9: Isolated Access History Maintenance Times (in seconds)

(*Monitor(1)* and *Monitor(2)*) less the execution time when simply maintaining concurrency information (*Concurrency*). These relative time increases indicate that as reader sets grow in size, the cost of detecting anomalies increases more rapidly for English-Hebrew labeling than for task recycling. Therefore, as larger reader sets are used the cost of the more complex task concurrency verification in the English-Hebrew algorithm will become increasingly more significant.

# 7 Concluding Remarks

This paper presents the problem of detecting access anomalies in parallel programs under two different models of concurrency: the pure fork-join model and a model which incorporates synchronous and asynchronous coordination. Several existing on-the-fly algorithms - merging and English-Hebrew labeling - and described and compared with a new algorithm based on task assignment to a partial order execution graph. Analytical comparisons of these three algorithms, both for average and worst case scenarios, are provided. Experimental data from four benchmark scientific programs is also presented for values of the parameters used in the analysis. Results obtained show that merging, which works well for programs with many concurrent reads, is not beneficial. The benchmark programs use data partitioning so extensively that over 80% of all variables never have more than two concurrent readers. Moreover, the size of access histories appears to be independent of the degree of parallelism within the program, so that overhead of monitoring using the two access history algorithms is effectively constant per variable access. Of the two access history algorithms, the primary advantage of task recycling over English-Hebrew labeling is that it needs much less storage for concurrency information. All metrics for comparing the three algorithms - purely analytical, analytic based on the measured parameter, and algorithmic implementation measurements - indicate that task recycling requires less space and computation time than the other two algorithms. If the benchmark programs are indicative of a wide class of parallel programs, the task recycling algorithm is an important improvement over existing technology.

There are several extensions to on-the-fly access anomaly detection, currently under investigation, to increase its functionality. The first extension involves the granularity of access control. In the current scheme, access to each primitive variable (e.g. an individual array

entry or structure element) is monitored; there is no mechanism for monitoring more abstract data constructs. Significant space savings are possible if functionally relate shared memory locations are encapsulated. Additionally, granularity in time should also be under user control. For example, it is sometimes desirable to encapsulate different partitions of an array as a single entity during different phases of an algorithm. A second extension is to differentiate between mutual exclusion and other coordination primitives. In previous work [6,14] mutual exclusion is modeled as a signal-event primitive. The result is that the concurrency graph is overly complex, detection is more expensive, and occasionally anomalies may be hidden. To this end, we introduce two new event types *mutex-read* and *mutex-write* that are used to record accesses in critical sections. Two *mutex* operations never conflict; however, mutex-write conflicts with both read and write and mutex-read conflicts with write. With this extension, programs with fork-join and mutual exclusion operations only may be modeled by the simpler series-parallel partial order execution graphs discussed in Section 3. Lastly, we are investigating the detection of more general race conditions by including in access histories the events which pertain to a given shared resource. Access anomalies are a special case of a broad class of time-sensitive behaviors. For this purpose an additional protocol, e.g. path expressions [5], is required to specify the correct ordering of events to that resource. The protocol used to verify the operational consistency of the operation, the partial order execution graph, and access histories are used to verify the temporal consistency of the operation.

**Acknowledgements**

We would like to thank Marc Snir and Larry Rudolph for their introduction to this problem and Ron Cytron, Allan Gottlieb, Bud Mishra and Ed Schonberg for their many helpful suggestions.

## References

[1] Todd R. Allen and David A. Padua. Debugging Fortran on a Shared Memory Machine. In *Proceedings of the International Conference on Parallel Processing*, pages 721–717, Aug 1987.

[2] Bill Applebe and Charles E. McDowell. Developing Multitasking Application Programs. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 94–102, Jan 1988.

[3] A. J. Bernstein. Program Analysis for Parallel Processing. *Transactions on Electronic Computers*, EC-15(5):757–762, October 1966.

[4] David Callahan and Jaspai Subhlok. Static Analysis of Low Level Synchronization. In *Proceedings on the SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 100–111, May 1988.

[5] R.H. Campbell and A.N. Haberman. *The Specification of Process Synchronization by Path Expressions.* Technical Report, Carnegie-Mellon University, December 1973.

[6] Jong-Deok Choi, Barton P. Miller, and Robert Netzer. *Techniques for Debugging Parallel Programs with Flowback Analysis.* Technical Report, University of Wisconson, Aug 1988.

[7] Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminancy in Parallel Programs. In *Proceedings on the SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[8] Allan Gottlieb. An Overview of the NYU Ultracomputer Project. In J.J. Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 25 – 95, Elsevier, 1988.

[9] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ Algorithm of Maximum Matching in Bipartite Graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), Jul 1978.

[11] Barton P. Miller and Jong-Deok Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings on the SIGPLAN Workshop on Parallel and Distributed Debugging*, May 1988.

[12] Itzhak Nudler and Larry Rudolph. Indeterminancy Considered Harmful. 1988.

[13] Itzhak Nudler and Larry Rudolph. Tools for the Efficient Development of Efficient Parallel Programs. In $1^{st}$ *Israeli Conference on Computer System Engineering*, 1988.

[14] Edith Schonberg. On-The-Fly Detection of Access Anomalies. In *Proceedings on the SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1989.

[15] Marc Snir. Private correspondence. 1988.

[16] Richard N. Taylor and Leon J. Osterweil. Anomaly Detection in Concurrent Software by Static Data Flow Analysis. *IEEE Transactions on Software Engineering*, SE-6(3), May 1980.

# A Proofs of Properties of Partial Order Execution Graph

**Definitions:**

- The terms $create_b$ and $term_b$ denote the creation and termination times for a block $b$.
- Let $V_1$ and $V_2$ be two disjoint sets of vertices of a partial order execution graph G = (V,E) such that each vertex in $V$ is in either $V_1$ or $V_2$, and all edges in E which have one end point in $V_1$ and one in $V_2$ are directed from $V_1$ to $V_2$. An *oriented cut* is defined to be the set block edges from vertices in $V_1$ to vertices in $V_2$.
- A *path cover* is defined to be a set of paths from the root to the leaf node which together traverse every block edge in the graph at least once.
- The *entry point* in the free task dag associated with an executing block $b$ is the child vertex of the vertex which is associated with $b$.
- A task in the free task dag is *reachable* by a block $b$ if the task is associated with a vertex which is on a path from the entry point of $b$ to the root of the dag.
- The term $avail_b$ is the number of tasks reachable by a block $b$.

**Theorem 1** *Two blocks $b_i$ and $b_j$ in a partial order execution graph G can execute in parallel iff $b_i$ is neither an ancestor nor a descendant of $b_j$ in G.*

**Proof.** $\Longrightarrow$ If a block $b_i$ is concurrent with $b_j$ then $b_i$ is neither an ancestor nor a descendent of $b_j$. Suppose, for a contradiction, that there is a block $b_i$ which is both concurrent with and an ancestor of another block $b_j$. Because block $b_j$ is a descendent of $b_i$, it can only be created after $b_i$ has terminated. Therefore, it must be the case that $term_{b_i} \leq create_{b_j}$. However, for two blocks to be concurrent it must be possible for both of their creation times to precede either termination times. Therefore, there is a contradiction.

$\Longleftarrow$ If a block $b_i$ is neither and ancestor nor descendant of another block $b_j$, then $b_i$ must be concurrent with $b_j$. Suppose, for a contradiction, that there are two blocks $b_i$ and $b_j$ such that $b_i$ is neither an ancestor nor descendent of $b_j$ and $b_i$ and $b_j$ are not concurrent. Define the *relatives* of a block to be all of its ancestors and itself. Let $a_i$ and $a_j$ be the closest non-common relative of $b_i$ and $b_j$ which are concurrent.

- If $b_i = a_i$ and $b_j = a_j$ then $b_i$ and $b_j$ are concurrent.
- If $b_i \neq a_i$ and $b_j \neq a_j$ we know that $a_i$ is not concurrent with any children of $a_j$ (because they are all closer relatives of $b_j$ than $a_j$). Therefore, $term_{a_i} \leq term_{a_j}$. Likewise, $term_{a_j} \leq term_{a_i}$. Hence, $term_{a_i} = term_{a_j}$. The only was to enforce this is if $a_i$ and $a_j$ synchronize which would make them relatives of both $b_i$ and $b_j$.
- If $b_i = a_i$ and $b_j \neq a_j$ (or vice verse), we know $term_{b_i} \leq term_{a_j}$. To enforce this, $b_i$ (or one of its descendents) must synchronize with $a_j$ (or one of its ancestors), and hence $b_i$ is an ancestor of $b_j$.

In all cases we get a contradiction and hence $b_i$ and $b_j$ can not have any non-common ancestors which are concurrent.

Let $a_0$ be the closest common relative of $b_i$ and $b_j$. We know that every two blocks have at least one common relative (the root block).

- If $a_0 = b_i$ (resp. $b_2$) then $b_i$ is an ancestor (resp. descendent) of $b_j$.

- If $a_0 \neq b_i$ and $a_0 \neq b_j$ block $a_0$ must create at least two children one which is a relative of $b_i$ and another which is a relative of $b_j$ (since there are no closer common relatives). However, these two children are concurrent and are hence non-common concurrent relatives of $b_i$ and $b_j$.

In all cases we have reached a contradiction and hence if $b_i$ and $b_j$ are concurrent, $b_i$ cannot be either an ancestor nor descendent of $b_j$. □

**Lemma 1** *A set of block edges are concurrent iff they are in an oriented cut.*

**Proof.** $\Longleftarrow$ Suppose, for a contradiction, that there are two edges $(v_i, a_0)$ and $(a_k, v_j)$ in an oriented cut and $(v_i, a_0)$ is an ancestor of $(a_k, v_j)$. From the definition of an oriented cut we know that $v_i$ and $a_k \in V_1$ and $a_0$ and $v_j \in V_2$. Consider all of the vertices $a_1$ ... $a_k$ on the path from $a_0$ to $v_j$. If $a_1 \in V_1$ the edge $(a_0, a_1)$ would go from $V_2$ to $V_1$; therefore, $a_1$ must be in $V_2$. Similarly, if $a_2 \in V_1$ the edge $(a_1, a_2)$ would go from $V_2$ to $V_1$; therefore, $a_2$ must be in $V_2$. By continuing this argument we deduce that $a_{k-1}$ must be in $V_2$. However, from our original assumption we know that $a_k$ is in $V_1$. Therefore, if two non-concurrent blocks are in the cut there must be an edge from $V_2$ to $V_1$ which is a contradiction.

$\Longrightarrow$ Suppose, for a contradiction, that there is a set of concurrent blocks $B$ which are not all in any oriented cut. There must be a subset $B'$ of $B$ such that all blocks in $B'$ are in at least one oriented cut $O$, but $B'$ is never in a cut with some other block $b \in B$. We know that $b$ must have either an ancestor or descendent edge $b'$ in $O$. If $b'$ is an ancestor of $b$, we know that there is no path from $b'$ or any descendent of $b'$ to any other edge in $O$. Therefore, we can move $b'$ and all edges on the path from $b'$ to $b$ to $V_1$ and make $b$ part of $O$. (A similar argument holds if $b'$ is a descendent of $b$). Because we can add $b$ to $O$, a contradiction has been reached. □

**Lemma 2** *All blocks in an oriented cut of size $C$ are traversed by at most one path in every path cover of size $C$.*

**Proof.** This follows directly from Lemma 1 and the fact that a path cannot traverse concurrent blocks. □

**Lemma 3** *The number of paths needed to cover a graph is equal to its maximum concurrency* T.

**Proof.** (By induction on $T$) : Base Case ($T = 1$): This graph is simply a linked list which can be covered with one path.

Induction: Suppose that the lemma holds for all graphs with $T < i$; prove that a graph with $T = i$ can be covered with $i$ paths. Create the leftmost path $L$ from the root to the leaf block and let $l$ be the set of blocks in $L$ which appear in cuts of size $i$. We know that $l$ contains exactly one edge from every directed cut of size $i$ in the graph. We modify $G$ to obtain a new partial order execution graph $G'$ with maximum concurrency $i - 1$ by deleting all block edges in $l$ as follows.

Consider the two end vertices, $v_p$ and $v_c$, of each block edge $e \in l$. By Lemma 2 there can never be more than one path through $e$ Therefore, if $v_p$ is a receiver vertex its associated coordination edge can be deleted and $v_p$ changed into a *null* vertex, (a vertex with only one parent and one child). Likewise, if $v_p$ is a synchronization vertex it is changed into a sender vertex. We know that $v_p$ cannot be a join vertex: otherwise, the parents of $e$ would be part of

a subgraph with maximum concurrency $> i$. By symmetric arguments, if $v_c$ is a sender vertex, we delete the coordination edge and change it to a null vertex; if it is a synchronization vertex we change it to a receiver, and we know that it cannot be a fork vertex. After making this transformation, every block edge in $l$ has a parent vertex which is either a sender, null, or fork vertex and a child vertex which is either a receiver, null, or join vertex.

Every maximal sequence of consecutive blocks $S$ containing only blocks in $l$ has the following properties:

- $S$ begins with either a fork or receiver vertex. If $S$ began with a null vertex $v$, the parent block of $v$ would be concurrent with the same set of blocks as $v$, would be in a directed cut of size $i$ and should therefore be part of sequence $S$.
- Likewise, $S$ ends with either a join or sender vertex.
- All internal vertices in $S$ are null vertices and hence $S$ has no internal coordination edges. Each internal vertex is the child of one edge in $l$ and therefore cannot be a join or sender. However, it is also the parent of an edge in $l$ and therefore cannot be either a receiver or fork vertex.

Each sequence $S$ in $l$ can be deleted from $G$ creating a new partial order execution graph $G'$ with maximum concurrency $i-1$. By our induction hypothesis, $G'$ can be covered with $i-1$ paths and therefore the lemma holds for all $i$. □

**Theorem 2** *The minimum number of tasks needed to perform a valid assignment to a partial order execution graph is equal to the maximum concurrency of the graph.*

**Proof.** By Lemma 1, the maximum concurrency $T$ of a graph is equal to the size of the maximum oriented cut of $G$. The minimum number of tasks needed to assign a graph is less than or equal to the size of the minimum path cover: all blocks on a path can be assigned to the same task because they are related by an ancestor/descendent relationship and by Theorem 1 are not concurrent; a block which is traversed by more than one path can be assigned to the task associated with any of the paths. From Lemma 3 we know that the number of paths needed to cover a graph is $\leq T$. Because each concurrent block needs a unique task, exactly $T$ tasks are needed to perform a valid optimal assignment. □

**Lemma 4** *A sequence of completed fork-join constructs results in a single vertex in the free task dag which always has a single child vertex.*

**Proof.** The vertex *ffree(f)* is defined to be the closest vertex with free tasks which is a descendent of the vertex $f$ associated with the fork operation of a fork-join pair. The following is a proof of induction by the depth of nesting $d$ of fork-join constructs and the length of the sequence $l$.

**Basis** ($d = 0$) : The sequence consists of simple fork-join sets, $f_1 \ldots f_l$. We prove by induction on the length of the sequence that the resultant dag consists of one vertex $v$ which has one child pointer to ffree($f_l$).

> **Basis** ($d = 0, l = 1$) : The sequence consists of a single fork-join pair, $f$. Each block created by the fork has ffree($f$) as its entry point. When each block terminates its entry point (e.g. ffree($f$)) is made a child of the join vertex and its associated vertex is deleted. If ffree($f$) is modified at any point in time, all entry points and the join vertex being created are modified to reference the new vertex associated with ffree($f$). The resultant subgraph is this single join vertex which has one child, ffree($f$).

**Induction** ($d = 0, l > 1$) : We assume the hypothesis holds for a sequence of length $l-1$, and prove that it holds for a sequence of length $l$. By our induction hypothesis we know that after assigning the first $l-1$ fork-join sets, the resulting dag consists of one vertex $v$ which has one child pointer to ffree($f_{l-1}$). There are two cases to consider:
I. $|f_l| \leq |f_1| \ldots |f_{l-1}|$. All blocks have $v$ as their entry points. When the last block frees its task, the new join vertex points to $v$ which has only one parent. Hence, $v$ is subsumed into the join vertex and the resultant dag consists of a single vertex with one pointer.
II. $|f_l| > max(|f_1| \ldots |f_{l-1}|)$. All blocks have their ffree($f_l$) as their entry points. All free tasks associated with the vertex $v$ are used and hence $v$ is collapsed and deleted. When all blocks terminate, the entry points are added to the $l^{th}$ join vertex, and the resultant dag consists of a single vertex with one child ffree($f_l$).

**Induction** ($d > 0$): Assume that for all sequences of vertices with maximum nesting $d-1$ the induction hypothesis holds; namely, the resultant dag consists of a single vertex with one pointer. We prove by induction on the length of the sequence it holds for a sequence of fork-join sets with maximum nesting $d$.

**Basis** ($d > 0, l = 1$) : The sequence consists of a single nested fork-join construct, $f$. If $f$ has nesting less than $v$ the induction hypothesis holds trivially. If $f$ has nesting $d$, then each of the parent blocks of the outermost join is the end of a sequence of fork-join constructs with maximal nesting of $d-1$. By our induction hypothesis, we know that the resultant subdag for each child consists of a single vertex $n_c$ which has one pointer to a to ffree($f$). When each blocks terminates, its entry point is made a child of the join vertex and thus, the resultant subgraph is this single join vertex which has one child ffree($f$).

**Induction** ($d > 0, l > 1$) : We assume the hypothesis holds for a sequence of length $l-1$, and prove that it holds for a sequence of length $l$. By our induction hypothesis we know that we can assign the first $l-1$ fork-join sets in the sequence and the resultant dag consists of one vertex, $v$, which has one child pointer to ffree($f_{l-1}$). If $f_l$ has maximum nesting less than $d$ then the induction hypothesis holds trivially. Otherwise, there are two cases which are analogous to the cases for $d = 0, l > 1$.

Hence, the lemma must hold for any depth of nesting of fork-join constructs in sequences of any length. □

**Lemma 5** *At all times the structure of the free task dag is a tree.*

**Proof.** From Lemma 4 we know that every completed fork-join construct corresponds to a single node. From the definition of the free task dag every open fork operation corresponds to a $f$-way branch in the dag. Thus, there is a vertex associated with each child of an open fork operation with the height of the free task dag bounded by the nesting of open fork operations. □

**Theorem 3** *The task assignment performed by the MRU algorithm is always optimal on series-parallel partial order execution graphs.*

**Proof.** We first note the following facts: (i) it is always optimal to assign all of the tasks from a parent vertex in the free task dag before assigning any from its child vertices (every block which can select from a vertex $v$ can also select from all of $v$'s children, but not visa verse), (ii) the selection among the tasks in a given free task set does not affect the optimality of the task assignment, and (iii) there always exists an optimal assignment. The MRU algorithm

always assigns all tasks associated with a parent before using any of those associated with a child vertex. Lemma 5 states the structure of the free task dag is a tree; therefore, once a parent runs out of free tasks there is no choice as to which child to use next. It follows from this and the above three facts that the task assignment performed is always optimal. □

**Theorem 4** *The size of the free task dag used by the MRU algorithm on series-parallel partial order execution graphs is bounded by* $O(T)$.

**Proof.** There can be at most $T$ vertices with non-empty free task lists (since there are at most $T$ unassigned tasks) and $T$ entry point vertices. From Lemma 5 we know that every vertex has exactly one child. Hence, the size of the free task dag is bounded by $O(T)$. □

**Theorem 5** *The amortized cost of assigning a task to a block for the MRU algorithm on series-parallel partial order execution graphs is* $O(N)$.

**Proof.** Consider the operations which must be performed for each block:

- Assignment: We create a vertex which is associated with the created block, make its entry point be the vertex associated with the fork operation, and use a task from the entry point.
- Collapse : If we use the last free task from the entry point during an assignment operation, we must update all vertices which point to this vertex.
- Free: We add the task to the free task set of the new vertex, make the child of the new vertex be the entry point of the block (if the new vertex's child is not set), and delete the vertex associated with the block.
- Subsume : If the entry point has a single parent vertex after a free operation, we perform a subsume operation by adding its free task set to the new vertex, and modifying the new vertex to point to the entry point's child.

We perform an assignment and possibly a collapse operation when a block is created; we perform a free and possibly a subsume operation when a block is terminated. The assignment, free and subsume operations perform $O(1)$ work; the collapse assignment performs work linear in the |child list| $\times$ |parent list|. We must show that the collapse assignment is performed sufficiently seldom to increase of overall amortized cost to $O(N)$.

From Lemma 5 we know that the |child list| is equal to 1, thus the work performed is linear in |parent list|. A parent is updated whenever one of its child vertices has all of its free lists deleted. This can only happen while the child is part of an open fork-join construct. Hence, each parent can be updated at most $N$ times where $N$ is the level of nesting of fork-join constructs. The number of parents in the dag is bounded by $B$ giving an amortized cost per block for task assignment of $O(N)$. □

**Theorem 6** *There is a cycle of length greater than two in a general partial order execution graph iff the blocks in the cycle are deadlocked.*

**Proof.** $\Longleftarrow$ Suppose, for a contradiction, that there is a cycle in the graph of block edges $b_1 \ldots b_n$ and the blocks are not deadlocked. There must exist some order of execution in which they all terminate. For $1 \leq i \leq n$ there is an edge from $b_{i+1}$ to $b_i$. Therefore, $b_i$ cannot begin before $b_{i+1}$ terminates and must terminate strictly after $b_{i+1}$ terminates. Moreover, there is an

edge from $b_1$ to $b_n$ and consequently that $term_{b_n} > term_{b_1}$. Therefore, $term_{b_1} > \ldots > term_{b_n}$ and $term_{b_n} > term_{b_1}$ which is a contradiction.

$\Longrightarrow$ Suppose, for a contradiction, that some blocks are deadlocked and there is no cycle in the partial order execution graph. There must exist at least one block $b_i$ in the deadlock whose ancestors $b_{i_1} \ldots b_{i_j}$ are not in the deadlock. At some point block $b_{i_1} \ldots b_{i_j}$ will terminate (since they are not deadlocked) and $b_i$ can terminate. Hence $b_i$ can not have been deadlocked which is a contradiction. □

**Theorem 7** *There exists an offline optimal task assignment algorithm for general partial order execution graphs which requires* $O(B^{2.5})$ *work.*

**Proof.** We construct a bipartite graph $G_B = (\{P, C\}, E)$ for a given partial order execution graph $G$ such that:

$|P| = |C|$ = the number of blocks in the partial order execution graph, and
$(p_i, c_j) \in E$ iff $p_i \in P$, $c_j \in C$ and block $b_j$ is a descendent of block $b_i$ in $G$.

A matching $M$ consists of $|M|$ pairs of vertices where $M_C$ denotes the set of end points in $M$ from vertex set $C$, $M_P$ denotes the set of end points from vertex set $P$ and each vertex in $M_C$ has a unique *partner* in $M_P$. Two properties follow from this definition:

1. Every matching $M$ of $G_B$ corresponds to a valid task assignment which uses $|C|$ - $|M|$ tasks. Each block vertex $\in M_C$ is assigned to the task associated with its partner block vertex in $M_P$ and each block $\notin M_C$ is assigned to a unique task. This assignment is valid because each task is assigned to a set of non-concurrent blocks and every block has a task assigned to it. Moreover, the number of tasks required for the assignment is equal to $|C| - |M|$.
2. Every valid task assignment which uses $T$ tasks has a corresponding matching $M$ of $G_B$ of size $|C| - |T|$. For each task $t$ add to $M$ every edge in $G_B$ which connects $p_{t_i}$ and $c_{t_{i+1}}$ ($t_j$ is the $j^{th}$ block assigned to task $t$). This is a matching because each block is assigned only one task; the only vertices in $C$ which are not included are the first blocks assigned to each task (e.g. $t_0$ for each task $t$) and hence $|M|$ is $|C| - |T|$.

It follows directly from 1 and 2 that by creating a maximum matching - which requires time $O(B^{2.5})$ [9] - we obtain an optimal task assignment. □

**Lemma 6** *There exists a partial order execution graph with maximum concurrency* $2^{i+1} - 1$ *such that at least* $(2^{i+1} - 1) + (2^{i-1} - 1)$ *tasks are required by any online valid task assignment.*

**Proof.** (By adversary argument): Consider the partial order execution graph in Figure 13; it has three branches each of which has a fork-join set of containing $2^i - 1$ parallel blocks. The set $X_i$ is a child of both $L_i$ and $R_i$. The maximum concurrency of this graph is therefore $2^{i+1} - 1$.

We will order the execution such that the blocks in $X_i$ are assigned before either block $l$ or $r$. The tasks last assigned to block $x$ and the blocks of $L_i$ and $R_i$ are reachable from the blocks in $X_i$. After the task assignment of the blocks of $X_i$, the number of tasks reachable from $l$ and $r$ is equal to the sum of the sizes of $L_i$ and $R_i$ less the tasks used from them during the assignment; namely, $avail_r + avail_l = 2 \times (2^i - 1) - (2^i - 2) = 2^i$. Therefore, either $avail_l$ or $avail_r$ must be less than or equal to $2^{i-1}$.

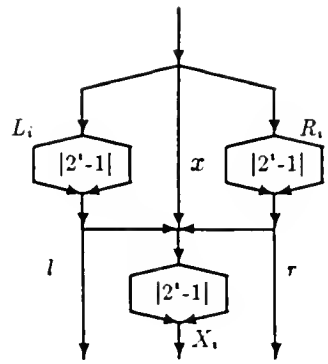



Figure 13: Subgraph at time $t$

If $avail_l \leq 2^{i-1}$, we create a subgraph of $l$ which is similar to the original graph except that each fork-join set contains $2^{i-1} - 1$ parallel blocks (shown in Figure 14a); a symmetric construction is used if $avail_r \leq 2^{i-1}$. This modification does not changed the maximum concurrency of the partial order execution graph graph. There are four groups of blocks which form maximally oriented cuts for each of the three branches of the graph (as in seen in Figure 14b); each group has maximum concurrency of $2^i - 1$. There are three oriented cuts of the entire graph which pass through more than one of these groups: the first contains the groups $A$ and $D$ and the solitary block $c$, the second contains $B$, $D$ and $c$, and the third contains $B$, $C$ and $d$. Therefore, the maximum concurrency of the graph remains $2^{i+1} - 1$.

The number of tasks needed for the task assignment of the subgraph is equal to its maximum concurrency $T'$. Hence, the number of *extra* tasks required is $T'$ less the number of tasks reachable from $l$; namely, it is at least $(2^i - 1) - (2^{i-1})$. Therefore, the total number of tasks required for a valid assignment to the entire graph is at least $(2^{i+1} - 1) + (2^{i-1} - 1)$. □

**Theorem 8** *A worst case lower bound on the number of tasks needed for any online task assignment algorithm on general partial order execution graphs is* $\frac{3T}{2} - log(T)$.

**Proof.** We start with the graph described in the proof of Lemma 6. The blocks in the subgraph have no reachable tasks and hence Lemma 6 can be applied recursively to this subgraph. If we start with an initial fork-join set size of $2^i - 1$ we can create $i - 1$ nested subgraphs; the fork-jon sets of the last subgraph consist of a single parallel block. This never increases the maximum concurrency $T$ of the graph, which is $2^{i+1} - 1$, and the number of extra tasks required is at least:

$$\sum_{j=1}^{i-1} 2^{i-j} - 1 = (2^i - 2) - (i - 1) = 2^i - (i + 1) \geq \frac{T}{2} - log(T)$$

Therefore, the total number of tasks needed to validly assign this partial order execution graph is at least $\frac{3T}{2} - log(T)$. □

**Theorem 9** *The size of the free task dag used by the MRU algorithm on general partial order execution graphs is bounded by* $O(T^2)$.

**Proof.** We first note that the only vertices which have more than one child must be leaf vertices. Vertices created from coordination operations are always leaf vertices and never have

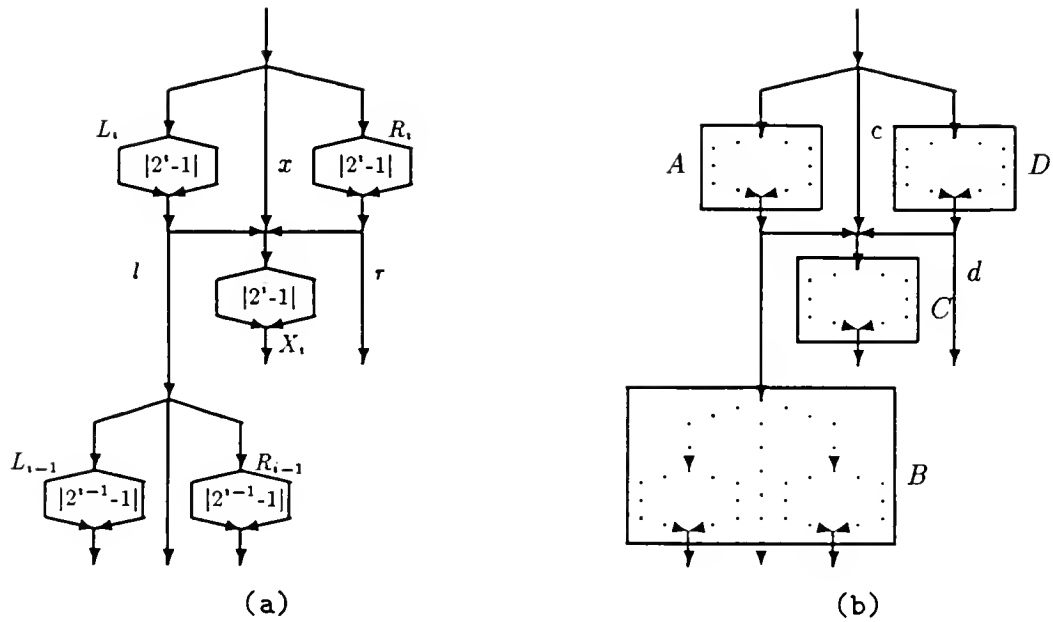



Figure 14: Subgraph at time $t+1$ if $avail_l \leq 2^{i-1}$

free tasks associated with them. Therefore, they are collapsed, rather than becoming the child of another vertex, when their associated block terminates. There can be at most $T$ vertices with non-empty free task lists (since there are at more $T$ unassigned tasks) and $T$ entry point vertices. Each entry point vertex can have as many as $T$ children; all other vertices have only 1 child. Therefore, there can be at most $O(T)$ vertices and $O(T^2)$ and hence, the size of the free task dag is bounded by $O(T^2)$. □

**Theorem 10** *The amortized cost of assigning a task to a block for the MRU algorithm on general partial order execution graphs is* $O(T)$.

**Proof.** As in the proof of Theorem 5, we must consider the operations which are be performed - assignment, collapse, free and subsume - as defined in the proof of Theorem 5: We perform a assignment and possibly a collapse operation when a block is created; we perform a free and possibly a subsume operation when a block is terminated. The subsume free and collapse assignment operation require $O(T)$ work only if they are performed on a leaf vertex. This can happen at most once per block. The remainder of the collapse and free operations are performed on internal vertices which continue to form a tree. Thus, the amount of work performed per block assignment and free is $O(T)$. □